RL-TR-96-167
Final Technical Report
October 1996

REAL-TIME PARALLEL SOFTWARE DESIGN CASE STUDY: IMPLEMENTATION OF THE RT-2DFFT BENCHMARK ON THE MASPAR MP-X ARCHITECTURE

The MITRE Corporation

David P. Koester and Joseph J. Rushanan

Rome Laboratory
Air Force Materiel Command
Rome, New York

RL-TR-96-167 has been reviewed and is approved for publication.

APPROVED: [illegible]
RALPH L. KOHLER, JR.
Project Engineer

FOR THE COMMANDER: [illegible]
DONALD W. HANSON, Director
Surveillance & Photonics Directorate

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE October 1996	3. REPORT TYPE AND DATES COVERED Final Mar 94 - Jun 95

4. TITLE AND SUBTITLE REAL-TIME PARALLEL SOFTWARE DESIGN CASE STUDY: IMPLEMENTATION OF THE RT-2DFFT BENCHMARK ON THE MASPAR MP-X ARCHITECTURE	5. FUNDING NUMBERS C - F19628-94-C-0001 PE - N/A PR - MOIE TA - 74 WU - 11
6. AUTHOR(S) David P. Koester and Joseph J. Rushanan	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) The MITRE Corporation Center for Air Force C3 Systems 202 Burlington Road Bedford, MA 01730-1420	8. PERFORMING ORGANIZATION REPORT NUMBER N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Laboratory/OCSS 26 Electronic Pky Rome, NY 13441-4514	10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-96-167
11. SUPPLEMENTARY NOTES RL Project Engineer: Ralph L. Kohler, Jr./OCSS/(315) 330-2016	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for Public Release; Distribution Unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The MITRE real-time embedded scalable high performance computing benchmarking concept was extended and tested by implementing the Real Time-Two Dimensional Fast Fourier Transform (RT-2DFFT) benchmark on the MasPar MP-X series of massively parallel processors (MPPs). The RT-2DFFT benchmark specifies a symmetric two-dimensional fast fourier transform (FFT) within a real-time software test bench. The test bench provides the realistic stimulus for the RT-2DFFT benchmark, including input-output (I/O) from/to on-board buffers. We developed a single RT-2DFFT implementation, heavily dependent on available library functions from MasPar, that can examine both benchmark latency specifications: latency equal to the period and latency greater than the period. Through the use of the MasPar RT-2DFFT benchmark implemenation, we show that the MasPar MPPs can read two-dimensional data set or input array from an I/O buffer, perform the two-dimensional FFT, and write the processed array out to an I/O buffer--all within the one second input array interarrival period specified in the benchmark. If latency is permitted to extend beyond one second, we show that it may be possible to reduce the machine size by processing sufficient multiple FFTs simultaneously, so that an entire row of a two-dimensional input array is assigned to a single processor. In this instance, the RT-2DFFT benchmark runs more efficiently, because communications overhead is minimized during both I/O and FFT processing.

14. SUBJECT TERMS Real Time-Two Dimensional Fast Fourier Transform, real-time embedded scalable high performance computing benchmarking, massively parallel processors, real-time software test bench.	15. NUMBER OF PAGES 108 16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

ABSTRACT

We extended and tested the MITRE real-time embedded scalable high performance computing benchmarking concept by implementing the RT_2DFFT benchmark on the MasPar MP-X series of massively parallel processors (MPPs). The RT_2DFFT benchmark specifies a symmetric two-dimensional fast Fourier transform (FFT) within a real-time software *test bench.* The test bench provides the realistic stimulus for the RT_2DFFT benchmark, including input/output (I/O) from/to onboard buffers. We developed a single RT_2DFFT implementation, heavily dependent on available library functions from MasPar, that can examine both benchmark latency specifications: latency equal to the period and latency greater than the period. Through the use of the MasPar RT_2DFFT benchmark implementation, we show that the MasPar MPPs can read a two-dimensional data set or input array from an I/O buffer, perform the two-dimensional FFT, and write the processed array out to an I/O buffer—all within the one second input array inter-arrival period specified in the benchmark. If latency is permitted to extend beyond one second, we show that it may be possible to reduce the machine size by processing sufficient multiple FFTs simultaneously, so that an entire row of a two-dimensional input array is assigned to a single processor. In this instance, the RT_2DFFT benchmark runs more efficiently, because communications overhead is minimized during both I/O and FFT processing.

ACKNOWLEDGMENTS

This work was supported by the United States Air Force Electronic Systems Center and performed under MITRE Mission Oriented Investigation and Experimentation (MOIE) Project 03957426 of contract F19628-94-C-0001, managed by Rome Laboratory/OCSS. We wish to thank the MasPar Computer Corporation and the Northeast Parallel Architectures Center (NPAC) at Syracuse University for the use of their MasPar machines. All product names, trademarks, and registered trademarks are the property of their respective holders.

The authors would also like to acknowledge the valuable assistance of Kelly T. Pickard of the MasPar Computer Corp during code development. We would also like to express our gratitude for the assistance of the systems support staff at NPAC—who configured vital input/output subsystems of the NPAC MasPar so that we could run our `RT_2DFFT` benchmark implementation.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

SECTION 1

INTRODUCTION

Previous internal MITRE efforts have developed a benchmarking methodology for real-time embedded scalable high performance computing (Games, 1996). This benchmarking methodology has been formally documented and proposed to the Defense Advanced Research Projects Agency (DARPA)/Information Technology Office (ITO) as a means to compare performance of the various parallel processing architectures being developed within DARPA/ITO's embedded computing program. The central focus of this benchmarking methodology is to determine the level of scalable massively parallel computer performance under realistic circumstances for a particular function. Much of the infrastructure and overhead of real-time processing is included in the benchmark implementations so that results are highly predictive of scalable massively parallel architecture performance. The RT_2DFFT benchmark was specified in (Games, 1996) as a test of the proposed methodology. We extended the MITRE real-time embedded scalable high performance computing benchmarking concept previously implemented on the Intel Paragon and implemented the RT_2DFFT benchmark on the MasPar MP-X series of massively parallel processors (MPPs).

The Intel Paragon is an example of a multicomputer with multiple independent computers connected with a mesh communications architecture. Flynn's taxonomy of computer architecture classifies this machine as a multiple-instruction stream, multiple-data stream (MIMD) architecture (Fox, 1988). In contrast, the MasPar MP-series is an example of an architecture that is distinctly different from the Intel Paragon and the MasPar MPP would be classified by the Flynn taxonomy as a single-instruction stream, multiple-data stream (SIMD) architecture. SIMD architectures have each processor execute the same instruction concurrently—with the option to software select any subset of the processors.

RT_2DFFT BENCHMARK

The RT_2DFFT benchmark specifies a simple symmetric two-dimensional fast Fourier transform (FFT) within a real-time software *test bench* (Games, 1996). This benchmark is applicable to synthetic aperture radar (SAR) image formation. SAR is among the premier near-term signal processing applications of scalable computing and is an appropriate initial focus for real-time embedded benchmarking activities. FFTs are *full-information* problems where every input value influences all output values. FFTs are also synchronous problems that can be processed in a tightly coupled *data parallel* manner. Parallel computer algorithms for FFTs are well documented in the literature, with FFT algorithms often being used as an example problem to describe entire algorithm classes for an architecture (Fox, 1988; Hwang, 1984; Quinn, 1987; Stone, 1987).

There are distinct limitations on the parallel programming paradigms that can be utilized with SIMD architectures (Fox, 1994). This fact is reflected in the limited implementation options for the `RT_2DFFT` benchmark on the MasPar MPPs. All efficient MasPar `RT_2DFFT` implementations must be synchronous and data parallel—compared to Intel Paragon `RT_2DFFT` implementations, which can include more complicated parallelization paradigms including pipelining in addition to data parallel paradigms.

To account for irregularities that affect timing such as overheads associated with non real-time operating systems, a benchmark is typically run iteratively for a long duration. In a hard real-time context, *worst case* performance is the metric of interest and must be within the stated real-time requirements.

TEST BENCH

The software test bench is an environment within which the real-time nature of applications—such as the `RT_2DFFT` benchmark—can be examined. The test bench provides the realistic stimulus for a benchmark and includes the infrastructure and overhead associated with a real-time implementation. This overhead includes data buffering and flow control. The test bench also includes the necessary software to collect and display performance statistics and to verify computational results. The test bench consists of a dedicated data source that provides data to the function under test and a data sink that collects the desired results. The source for an actual embedded system would be external and stream data into a buffer. Results collected at the data sink might be displayed in real-time on a graphical front-end workstation or on a graphical display attached to a frame-buffer. External input/output (I/O) is often the *last frontier* for massively parallel systems, so the initial implementation of the test bench does not attempt to interface with external data sources, rather internal buffers are used for the data source and data sink.

Due to differences in underlying architectures, separate test bench implementations have been developed for the Intel Paragon (Brown, 1994) and the MasPar MPPs. The primary differences between test bench implementations are a result of the hardware available to support the data source and sink on these machines. The MasPar test bench implementation requires a custom developed software architecture to maximize real-time application performance by minimizing interaction with the UNIX-based front-end processor.

HISTORY

The RT_2DFFT benchmark has evolved from previous work that describe implementations of real-time benchmarking on the Intel Paragon (Brown, 1994; Brown, 1995). Initial work on a parallel implementation of a two-dimensional FFT benchmark for the Intel Paragon, based on a pipelining programming paradigm, is discussed in (Brown, 1994). The FFT implementation in that paper capitalizes on the capability of a MIMD architecture to run different code on each processor to implement a parallel algorithm as pipelined meta-problems. A second application benchmarked within the real-time embedded test bench on the Intel Paragon is a SAR benchmark application (Brown, 1995). This work is important because it illustrates the validity of the test bench concept as separate functions were developed and individually tested, and then integrated into a functional real-time algorithm. As long as individual functions had predictable performance, they could be combined into the larger SAR application with predictable results.

Another real-time Department of Defense application using the MasPar MPP is the Theater Missile Defense Ground Based Radar (TMD-GBR). The MasPar MP-2 MPP system has been selected as the signal processor for the TMD-GBR system being developed by Raytheon Co., Lexington, MA., for the US Army Ground Based Radar Project Office in Huntsville, AL. The TMD-GBR is the sensor for the new Theater High Altitude Area Defense (THAAD) missile defense system being developed to protect friendly troops and population centers from tactical ballistic missiles similar to those fired by Iraq during Operation Desert Storm. This contract award marks the first time that commercial off-the-shelf (COTS) MPPs have been used in a large-scale, mission-critical, field-deployable system. The MasPar MP-2 was selected after rigorous competition because of its I/O capabilities, which provide both high bandwidth and low latency.

SUMMARY OF RESULTS

We developed a single RT_2DFFT implementation that extensively uses available library functions from MasPar and that can examine both benchmark latency specifications: latency equal to the period and latency greater than the period. Through the use of the RT_2DFFT benchmark implementation, we show that the MasPar MPPs can read a two-dimensional data set or input array from an I/O buffer, perform the two-dimensional FFT, and write the processed array out to an I/O buffer—all within the one second input array inter-arrival time specified in the benchmark—for 256 × 256, 512 × 512, and 1K × 1K single precision complex input arrays. If latency is permitted to extend beyond one second, we show that it may be possible to reduce the machine size by processing sufficient multiple FFTs simultaneously, so that an entire row of a two-dimensional input array is assigned to a single processor. In this instance, the RT_2DFFT benchmark runs more efficiently, because communications overhead is minimized during both I/O and FFT processing.

The smallest machine capable of meeting the timing specification for the latency-equal-period case varied from 1K to 8K for the MP-1 while a 1K MP-2 was able to process each of the three input array sizes successfully. We were in fact unable to process a 2K × 2K input array in the required one second inter-arrival period on either of the MasPar machines that we used to test our RT_2DFFT implementation; however, we anticipate that we should be able to process a 2K × 2K input array on an 8K processor MP-2 within the one second inter-arrival time requirement.

We identified constraints on maximum problem size for two-dimensional applications with single precision complex data array elements—some constraints are due to the hardware architecture and some constraints are due to the computational capabilities. Constraints on maximum PE memory impose a potential limitation on input array sizes of 8K × 8K rows and columns. Furthermore, maximum IORAM memory constraints reduce the maximum input array size to 4K × 4K rows and columns, and maximum I/O subsystem bandwidth constraints further reduce the maximum input array size to only 2K × 2K rows and columns.

Processing input arrays larger than 2K × 2K would not be possible on either the MP-1 or MP-2 even if I/O subsystem bandwidth limitations are overcome, assuming that we maintain the one second period requirement. We found that as the number of rows in the input array are doubled, the number of processors required to meet a constant timing requirement must increase by at least a factor of four. While this is not unexpected when processing two-dimensional input arrays for a two-dimensional FFT application, present MasPar processors are limited to only 16K processors and consequently growth to larger input arrays would be limited by the maximum size of the PE array. Given the trends apparent in the data, at least a 32K processor machine or a system speedup of a factor of four would be required to handle a 4K $\times$ 4K input array.

It should be emphasized that all these results apply to the `RT_2DFFT` benchmark specification given in appendix A. This benchmark specification maintains a one second inter-arrival period independent of input array size. As such, it represents a substantial real-time test of the underlying hardware and system software as the input array size is increased. Any actual application that would require such two-dimensional FFT processing, e.g., SAR, could have a longer inter-arrival time and specific latency requirements. The parametric benchmarking techniques and infrastructure described in this report could be easily adapted to assess the suitability of the MasPar MP-X architecture for a particular target application once the actual real-time requirements are specified.

ORGANIZATION OF THIS REPORT

This paper is organized as follows. In section 2, we provide some general information on some fundamental concepts used in this paper. Material presented in this section expands upon the `RT_2DFFT` benchmark implementation guidelines presented in appendix A. In section 3, we describe the MasPar MP-X series hardware and software architecture. In section 4, we describe the software and hardware architectures of the MasPar test bench. In section 5, we describe the FFT implementation selected for the `RT_2DFFT` benchmark for this machine. We present a detailed discussion on the implementation choices and the logic for the selection of our `RT_2DFFT` benchmark software architecture. We have made extensive use of MasPar library routines in the implementation, and in appendix B, we present details on the utilization of supplied routines that move data within parallel data structures.

In section 6, we discuss the practical constraints on the `RT_2DFFT` benchmark implementation imposed by available MasPar hardware configurations. We present empirical performance results for the MasPar `RT_2DFFT` implementation in section 7, and we present tabular collections of empirical performance data in appendices C and D, respectively for the MP-1 and MP-2. Our conclusions on the RT_2DFFT benchmark results and the applicability of the MasPar MPP architecture for real-time applications are presented in section 8.

SECTION 2

FUNDAMENTAL CONCEPTS

The MITRE benchmarking methodology for real-time embedded scalable high performance computing has been published as reference (Games, 1996). The `RT_2DFFT` benchmark was proposed to test the methodology and is reprinted in appendix A. In this section, we highlight some fundamental concepts from that specification that are critical to developing the `RT_2DFFT` benchmark for the MasPar MPPs. In particular, we discuss the real-time embedded processing and the `RT_2DFFT` benchmark, the difficulties with external I/O, and the manner with which we software-select the MasPar processor element (PE) array size.

REAL-TIME EMBEDDED PROCESSING

This paper examines a two-dimensional algorithm on a two-dimensional input *array* or *matrix*—symbolically referred to as A. In particular, we are implementing the `RT_2DFFT` benchmark, a symmetric two-dimensional fast Fourier transform. The objective is to find the smallest MasPar machine size that can meet benchmark timing requirements for various input array sizes. The two-dimensional input array size is described as a function of height and width, where array sizes are specified by the number of rows and columns—$m \times n$.

For the `RT_2DFFT` benchmark, all input arrays are square and limited to sizes of powers of two. For all potential input array sizes under test, all processors will be utilized in our data parallel, synchronous implementation for the `RT_2DFFT` benchmark, because both MasPar machine sizes and input array sizes are powers of two. An input array is identified by its dimensions—a 512 $\times$ 512 input array has 512 rows and columns. Matrices with 1024 rows and columns will be described as a 1K $\times$ 1K array, and those arrays larger than 1K $\times$ 1K will also be referenced using a similar notation—2K $\times$ 2K, 4K $\times$ 4K, etc.—where K always equals $2^{10} = 1024$.

Let us define a problem $\mathcal{P}$ as performing a two-dimensional FFT on a single input array. If the inputs for the problem become available to the computer at starting time t_s and the solution is completed at time t_c, then the processing *latency* for problem $\mathcal{P}$ is $(t_c - t_s)$. For the `RT_2DFFT` benchmark, we are assuming that there is a temporal stream of problem instances $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_i, \ldots$ with a fixed inter-arrival time τ, referred to as the problem inter-arrival time or *period.*

Consequently, input arrays arrive in the sequence $A_1, A_2, \ldots, A_i, \ldots$ with the fixed inter-arrival time τ. Corresponding processed output arrays are denoted by $Z_1, Z_2, \ldots, Z_i, \ldots$.

The RT_2DFFT benchmark stipulates two processing latency scenarios:

Case 1: latency equal to the period,

Case 2: latency greater than the period.

For the initial RT_2DFFT benchmark, the period is specified as one second. Other benchmarks for actual applications may specify other inter-arrival times. Video, for example, has new frames of data arriving every $\frac{1}{30}$th of a second, the *refresh rate.*

The *latency-equals-period* case would relate to those embedded applications where processing must be handled as quickly as data are generated by sensors due to the requirement for time-critical responses. Other embedded applications require only that the processed output must be generated at a rate that sustains the real-time rate of the input. For both latency cases, processed output must be generated at a rate that sustains the real-time rate of the input—the time between successive processed outputs must be less than or equal to τ. The *latency-greater-than-period* case induces a delay in the real-time output. For pipelined MIMD implementations of the RT_2DFFT benchmark, each problem instance yields a processed array (after the pipeline is filled). In contrast, for this latency case, our SIMD RT_2DFFT implementation collects multiple input arrays and processes them simultaneously.

Using notation for problems from above, multiple problems would be aggregated and processed simultaneously in parallel. If b consecutive problems are blocked together for the SIMD implementation latency-greater-than-period case, the problem definition can be stated as $[\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_b], [\mathcal{P}_{b+1}, \mathcal{P}_{b+2}, \ldots, \mathcal{P}_{2\cdot b}], \ldots$. This can be redefined as $\tilde{\mathcal{P}}_1, \tilde{\mathcal{P}}_2, \ldots, \tilde{\mathcal{P}}_k, \ldots$, where $\tilde{\mathcal{P}}_k$ represents the problem statement for a block of input data. Using similar notation as above, input for this case would be blocked together for b consecutive input arrays, $[A_1, A_2, \ldots, A_b], [A_{b+1}, A_{b+2}, \ldots, A_{2\cdot b}] \ldots$, and the output would be in the form $[Z_1, Z_2, \ldots, Z_b], [Z_{b+1}, Z_{b+2}, \ldots, Z_{2\cdot b}] \ldots$. The definition of block processing latency here is similar to the single problem case, except it is assumed that the starting time t_s occurs when a block of data is available for processing and the time t_c occurs when processing is completed on a block of data. In our SIMD implementation, only one block of problems can be processed at a time and so to maintain an input period of τ, the block processing latency cannot exceed $(\tau \cdot b)$. Equivalently, the normalized quantity $\frac{(t_c - t_s)}{b}$ will be used later to illustrate performance improvements for processing multiple input arrays simultaneously under the assumption of the second unrestricted latency case. This quantity corresponds to the measured period of a single problem instance.

To ensure that processed arrays are delivered to the display device correctly, the processed arrays within a block must be displayed at the input rate τ. Reinstating the timing by pacing the output of the processed arrays during display is referred to as *recovering clock.*

EXTERNAL I/O

The MITRE benchmarking methodology for real-time embedded scalable high performance computing does not specifically address external I/O, due to the complexities of this problem. External I/O is often the *last frontier* for massively parallel systems, so initial implementations of the test bench do not attempt to interface with external data sources. Resources internal to the MPP are used instead for the data source and data sink.

To illustrate these complexities, we briefly examine the networking requirements to connect an external data source to an MPP. The input arrays used in this version of the test bench are two-dimensional, with each element being a single-precision complex value. Stored in binary form, each array element requires eight bytes or 64 bits of information. In general, networking requirements expressed in bits-per-second are of order $O(n^2 \cdot \beta)$, where n is the input array dimension and β is the input array element size expressed in bits-per-element. Networking requirements are presented in Table 1 for representative input array sizes used in this analysis. At the input array arrival rate of only a single array-per-second, network throughput requirements for complex input arrays with dimensions greater than 256×256 rows/columns are so large that high-performance networking capabilities beyond simple 10 megabit-per-second Ethernet networks will be required to handle the throughput. Networking technologies capable of supporting these data rates include Asynchronous Transfer Mode (ATM), Fibre Channel, and the High Performance Parallel Interface (HiPPI) (Koester, 1994). At video refresh rates of 30 input arrays-per-second, the smallest input array will require high-performance networking technology for uncompressed arrays, while larger input arrays will re-

Table 1. Network Throughput Requirements for External I/O

	Data Throughput Rates	
Input Array Size n	1 array-per-second	30 arrays-per-second
256	4 megabits-per-second	120 megabits-per-second
512	16 megabits-per-second	480 megabits-per-second
1024	64 megabits-per-second	1920 megabits-per-second
2048	256 megabits-per-second	7680 megabits-per-second

quire networking capabilities well beyond any networking throughputs available today.

FINDING THE SMALLEST MACHINE

The real-time embedded benchmarking methodology has been developed explicitly to consider that the high-performance computer technology under test may be embedded into military equipment and command and control (C^2) platforms—where size, weight, and power consumption can be critical factors. Consequently, we are interested in finding the smallest machine that can sustain the processing requirements. It is critical that we can adjust machine size as we run the `RT_2DFFT` benchmark.

We have been able to capitalize on supplied software from MasPar to *software adjust* the size of the PE array to run the `RT_2DFFT` benchmark on machines of various simulated sizes. Before running the `RT_2DFFT` benchmark, we can specify the PE array configuration with the `mpswopt` command, which permits the user to select any simulated machine size up to and including the maximum size of the actual machine. MasPar MPPs always have numbers of processors equal to powers of two with the smallest machine having 1024 or 1K processors. MasPar MPPs are available with the following PEs array configurations:

- 2^{10} or 1K processors arranged with 32 rows $\times$ 32 columns,

- 2^{11} or 2K processors arranged with 32 rows $\times$ 64 columns,
- 2^{12} or 4K processors arranged with 64 rows $\times$ 64 columns,
- 2^{13} or 8K processors arranged with 64 rows $\times$ 128 columns,
- 2^{14} or 16K processors arranged with 128 rows $\times$ 128 columns.

By software adjusting the PE array size, we can examine various sized MasPar computers, without requiring hardware reconfigurations of those machines available to run the `RT_2DFFT` benchmark. Throughout this paper, all references to MasPar machine size will specify the number of processors with the short-hand notation "K"—where K always equals $2^{10} = 1024$.

SECTION 3

MASPAR DESCRIPTION

The MasPar MP-X series of MPPs combines a scalable architecture with respect to the number of processing elements, system memory, and system communications bandwidth, with a *reduced instruction set computer (RISC)-like* instruction set designed to leverage optimizing compiler technology, adherence to industry standard floating-point operations, and a suitability to very large scale integration (VLSI) implementation (Blank, 1990; MasPar Computer Corp., 1992). According to Flynn's taxonomy, the MasPar MPP architecture would be described as single instruction stream, multiple data stream or SIMD architecture. The basic MasPar MPP architecture has been relatively unchanged since its introduction in 1990. Scalability, leveraging the best computer science technologies, adherence to standards, and cost effective manufacturing techniques have contributed to the unusually long lifespan of the MasPar MP-X series MPP technology.

MASPAR SYSTEM OVERVIEW

The MasPar MPP system has five major subsystems that are described briefly below (Blank, 1990).

UNIX Front-End (UFE): The UNIX front-end provides a standards-based interface to the massively parallel PE array. The UFE provides all user interfaces to the MPP, including slow-speed networking interfaces, standards-based networking software, and a UNIX-based operating system.

Processor Element (PE) Array: The PE array is the massively parallel computation engine of the MasPar. The number of PEs can scale from 1K ($1024 = 2^{10}$) to 16K ($16,384 = 2^{14}$) processors.

Array Control Unit (ACU): The ACU performs two functions:

- The ACU controls the PE array by broadcasting all instructions to the massively parallel processor array.

- The ACU also functions as an independent processor and calculates all serial portions of MasPar Parallel Application Language (MPL) programs (MasPar Computer Corp., 1993).

Interprocessor Communications: Key to any parallel processor is the interprocessor communications structure. The MasPar MPP has three interprocessor communications capabilities for the PE array:

- A two-dimensional mesh network for communications with neighboring processors.

- A global router network for random processor-to-processor communications using a circuit-switched, hierarchical crossbar communications network. This high-performance router-based communications network can also provide high-performance links to the Input/Output (I/O) subsystem, because the last stage of the network connects to an I/O buffer, the IORAM.

- Two global buses: one bus for broadcasting instructions and data from the ACU to the PE array, and a second bus for receiving consolidated status responses from the PE array at the ACU.

Input/Output (I/O) Subsystem: The I/O subsystem provides high speed I/O with a channel style architecture that permits overlapped computation and communications. The MasPar architecture has been designed to support multiple gigabit-per-second HiPPI network interfaces and with a bus throughput within the I/O subsystem to support those high throughput rates. Input and output data are stored in a large random access buffer—the IORAM—that is connected to the PE array via the global router network.

The MasPar MPP has two separate instruction steams (the UFE and the ACU), and four separate memory structures for storing user data (the UFE, IORAM, the ACU, and the PE array). All PE instructions are stored in the ACU, and are broadcast to the PEs—thus the classification as a SIMD architecture. Due to the separate instruction streams, two programming approaches are possible and supported by MasPar:

- When developing application codes using only MPL, the programmer develops one application code that is automatically distributed between both the UFE and the ACU, and user data is partitioned across the UFE, ACU, and PE array. All interprocess communications is automatically handled by the compiler.

- When developing application codes for the UFE in the C programming language and the ACU/PE array combination in MPL, the programmer develops separate application codes for both locations and the programmer must explicitly partition the user data across the UFE, ACU, and PE array. All interprocess communications between the UFE and the MPP back-end must be explicitly handled when developing the programs.

Either programming approach can utilize synchronous or asynchronous interaction models. The synchronous programming model utilizes only the UFE or the ACU/PE array combination at any instant and a Remote Procedure Call (RPC) convention provides a straight-forward interface between the two separate processes. The asynchronous model, on the other hand, utilizes a `FORK/JOIN` model to permit the UFE and the ACU/PE combination to operate concurrently.

DESCRIPTIONS OF THE SPECIFIC MP-X MACHINES

We tested our `RT_2DFFT` implementation on two different MasPar platforms: a 16K MP-1 at the Northeast Parallel Architectures Center (NPAC) at Syracuse University and a 4K MP-2 at the MasPar Corporation office in Framingham, MA. The primary differences in these two generations of compatible systems are the processor performance capabilities. While the integer performance of the processor has improved by greater than a factor of two and floating point performance of the processor has improved by greater than a factor of five, the performance of the communications networks has remained essentially unchanged in the new architecture. Table 2 illustrates the differences in peak integer and floating point performance for the MP-1 and MP-2. In this table, peak integer performance is labeled as millions of instructions per second (MIPS), and peak floating point performance is labeled as Mflop/s.

Specifications of the machines used to implement the benchmark are presented in Table 3. In this table, we indicate the UFE machine type and its operating system (OS), the size of the PE array, and the available amount of each of four memory types. The amount of IORAM is shown as the amount available as we

Table 2. **MasPar Product Family**

System Generation	Number of Processors	MIPS	Mflop/s
MP-1	1K	1,600	75
	2K	3,200	150
	4K	6,400	300
	8K	13,000	600
	16K	26,000	1,200
MP-2	1K	4,250	400
	2K	8,500	800
	4K	17,000	1,600
	8K	34,000	3,200
	16K	68,000	6,300

ran the `RT_2DFFT` benchmark; in the case of the MP-1 the total IORAM was larger, but system configuration limited the amount available to be used as a data source/sink buffer. We were able to software reconfigure each machine to a smaller size, so our benchmark was run on MP-1 machines of size 1K to 16K and on MP-2 machines of sizes 1K to 4K.

We encountered another performance improvement in MasPar MPP models, although it is related to the date-of-manufacture rather than to the system model. In later machines, the UFE workstation has been upgraded. Earlier MP-1s and MP-2s used a DECstation 5000 workstation running ULTRIX as the operating system for the UFE. The UFE for later deliveries of MP-Xs has been upgraded to a Digital Equipment Corporation (DEC) Alpha workstation running Digital UNIX (formerly known as OSF/1), a variant of UNIX. The NPAC MP-1 has a DECstation UFE, while the MP-2 model has a DEC Alpha UFE. There is a significant performance improvement with the more capable Alpha processor and the Digital UNIX operating system.

Table 3. Descriptions of the MP-1 and MP-2 Utilized in this Report

	MP-1	MP-2
Front-End	DECstation	Alpha Workstation
FE OS	Ultrix V4.3	Digital UNIX (OSF)
PE Array	$16K = 128 \times 128$	$4K = 64 \times 64$
ACU IMEM	1 Mbytes	4 Mbytes
ACU CMEM	1 Mbytes	512 Kbytes
PE MEM	64 Kbytes	64 Kbytes
IORAM Size	32 Mbytes	128 Mbytes

The MP-2 was connected to an isolated network, where often we were the sole user on the system. On the other hand, the MP-1 was connected to the Internet; so, even if our `RT_2DFFT` benchmark was the only process running on the ACU/PE array, there was the possibility that another user could execute a process on the UFE and potentially affect the system clock calls and increase timing variability.

SECTION 4

MASPAR TEST BENCH IMPLEMENTATION

In this section, we describe the hardware and software architectures of the MasPar test bench. In section 5, we examine the implementation details of the RT_2DFFT benchmark FFT algorithm.

The software test bench provides the realistic stimulus for a benchmark and includes the infrastructure and overhead associated with a real-time implementation. This overhead includes data buffering and flow control. The test bench also includes the necessary software to collect and display performance statistics and to verify computational results. The test bench includes a dedicated data source that provides data to the function under test and a data sink that collects the desired results. The data source and sink must be separate from the ACU and PE array. The source for an actual embedded system would be external and would stream data into a buffer. Results collected at the data sink might be displayed in real-time on a graphical front-end workstation or on a graphical display attached to a frame-buffer.

SOFTWARE ARCHITECTURE

The MasPar test bench implementation requires a custom developed software architecture, the development of which has been driven by the requirements to implement the data source and data sink buffers. Buffering data in the IORAM requires access to both the UFE and the ACU/PE array combination, which in turn requires that the software be written partially in C for the UFE and the remainder in MPL for the ACU/PE array combination. Initialization of the I/O subsystem is only accessible from the UFE and cannot be invoked from an MPL program. As a result, a specialized software architecture was required where the main program, written in the C programming language, on the UFE calls the real-time test bench as an external function. At present, a synchronous interaction model is utilized. If the complete I/O structure is implemented—unprocessed arrays coming in over the HiPPI network and processed arrays being displayed via the frame buffer—the asynchronous interaction model will be required as the UFE controls the I/O processes concurrently while the ACU/PE array processes the computational application.

With this architecture, special considerations are required to maximize real-time application performance by minimizing interaction with the UFE. To ensure optimal performance, the real-time computing tasks must execute as a contiguous program on the PE array, while being controlled by the ACU. Running the real-time processing from the ACU ensures that disruptions due to communications from the UFE to the ACU via the slow virtual memory expansion (VME) bus are eliminated. Running the real-time processing from the ACU also isolates the UNIX operating system on the UFE from the repetitive processing.

The resulting test bench software architecture is presented in Figure 1. This figure clearly illustrates the relationship between the application specific module and the test bench or software test environment. The application specific module for the `RT_2DFFT` benchmark is a two-dimensional FFT. The test bench provides all the infrastructure within which to run the computational benchmark. The test bench provides software simulation of the I/O, post-processing, and verification of results. The test bench architecture requires that the command line parameter list be transferred to the control program running on the ACU so that the repetitive processing loop can be run from this MPL program.

HARDWARE ARCHITECTURE

In Figure 2, we present a diagram of the MasPar MPP that includes the program data flow as we run the test bench. This diagram illustrates that the test bench requires access to many of the subsystems within the MasPar. The test bench main program is run on the UFE, which distributes the command-line parameter list to the benchmark software running on the ACU and the PE array. Test input arrays can be read from disk into the UFE and can be transferred to the IORAM after this memory has been initialized by support functions running on the ACU.

Figure 2 includes the MasPar I/O subsystems that bring in external inputs and display the output on an internal frame buffer (MasPar Computer Corp., 1994). At present, we are not required to include external I/O in the `RT_2DFFT` benchmark; nevertheless, we include a complete hardware description of the I/O subsystem to illustrate that external I/O hardware is readily available for MasPar MPPs. To be fully operational, a real-time program must consider the flow of data in from an external HiPPI-connected source and out to a display connected to a frame buffer. In the present version of the test bench, we simulate I/O within the MasPar by implementing parallel data transfers between the PE array and

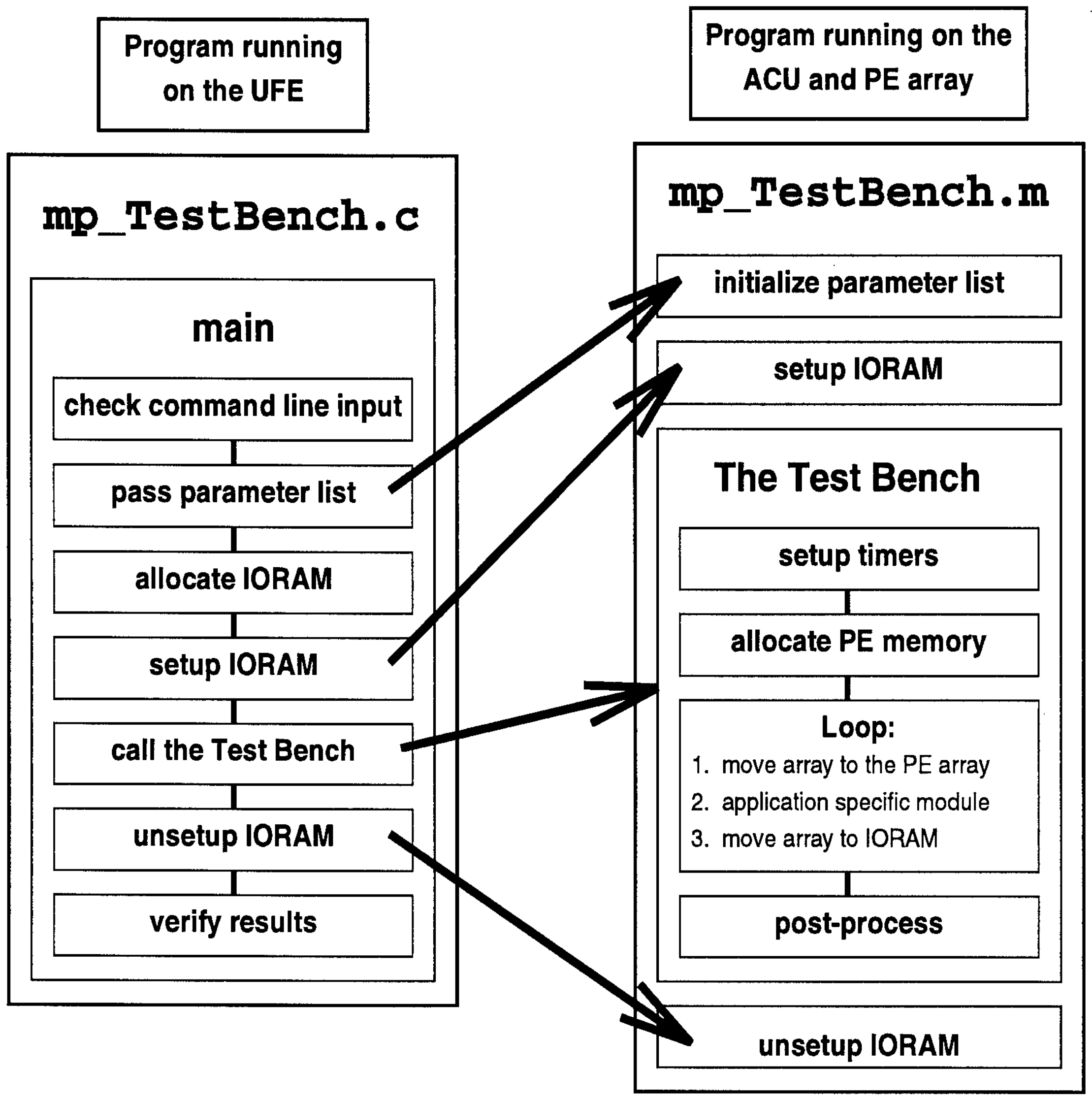


Figure 1. MasPar Test Bench Software Architecture

the IORAM—where the IORAM is both the data source and sink buffers. Later programs that fully implement data movement from the HiPPI interface to the IORAM or from the IORAM to the frame buffer would also utilize the IORAM to double buffer the data while the ACU/PE array is performing calculations on the data array. Further research is required to validate the real-time characteristics

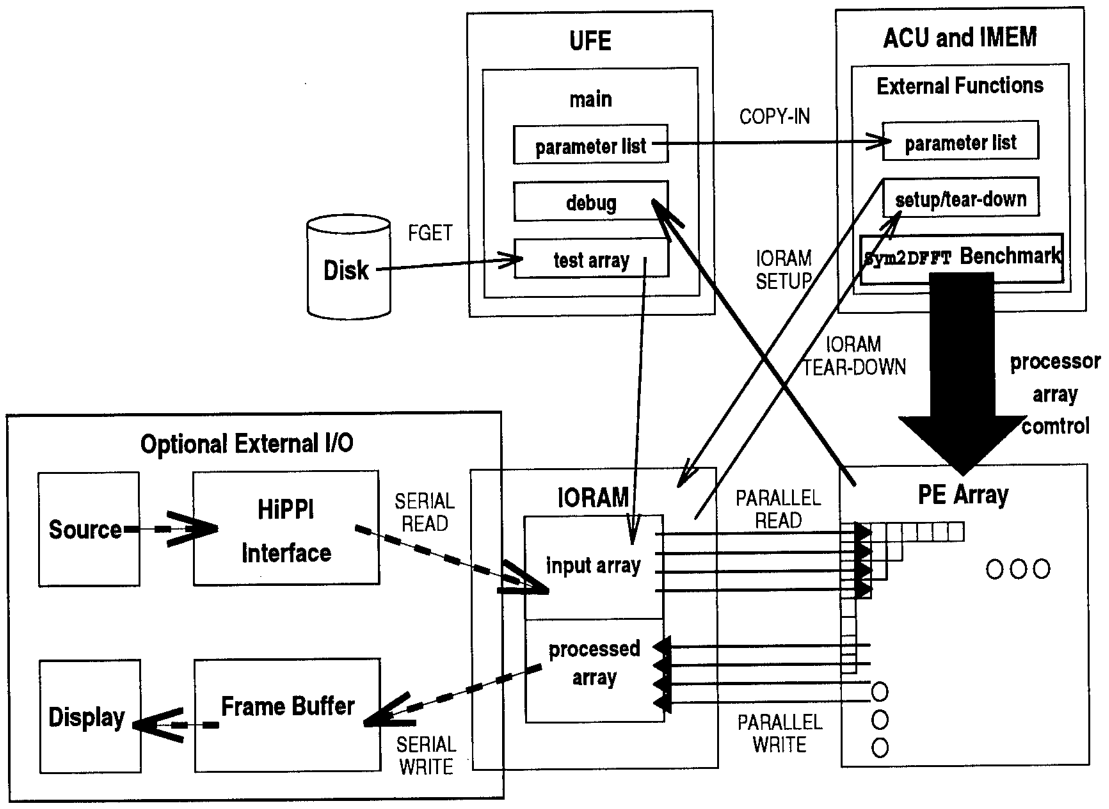


Figure 2. MasPar Test Bench Hardware Architecture

of asynchronously transferring data between the IORAM, HiPPI interface, and frame buffer.

The most important data flow within the program is the highly-parallel data transfers between the IORAM and the PE array. These read/write operations occur synchronously, in parallel, with data sent to processors using software programmable offsets. MPL provides the capability for *plural* variables, where a plural variable is allocated space on each processor in the PE array and is calculated in a *data parallel* manner. The values of plural variables generally are related to the spatial nature of the PE array. Read/write offsets are plural variables—as a result, any parallel read/write operation can be implemented.

A single library function call synchronously implements highly parallel data transfers between processors. These transfers are the first and last steps in each loop cycle within the test bench—the loop is repeated iteratively to gather timing statistics. Due to the synchronous nature of the SIMD PE array, it is impossible to double buffer this data transfer operation to hide the communications costs.

The last significant data flow within the test bench is reserved for debugging. The MasPar has excellent libraries of support software and we used the X-Windows-based display software to generate pseudo-images representing memory state throughout the PE array (MasPar Computer Corp., 1992). Consequently, while developing and validating the test bench, we viewed representations of both input and processed data arrays to validate algorithms and I/O procedures.

SECTION 5

MASPAR RT_2DFFT BENCHMARK IMPLEMENTATION

In this section, we examine the implementation details of the FFT algorithm used in the RT_2DFFT benchmark. We start by describing the parallelization alternatives for the MasPar SIMD architecture. Next, we discuss the FFT implementation options for the MasPar with a detailed examination of the FFT function from the MasPar mathematics library. Lastly, we describe in detail the mapping of input data matrices to the PE array.

PARALLELIZATION ALTERNATIVES

The SIMD architecture of the MasPar machines limits the parallelization options for the RT_2DFFT benchmark. Since each processing element in the PE array must execute the same instruction—with the option to software select a subset of the processors—the PE array cannot efficiently work simultaneously on more than a single data array unless the processing is synchronous. Consequently, we cannot efficiently use the PE array in a pipeline implementation, where one portion of the PE array processes a segment of one input array while another PE array portion processes a different segment of another input array. To be efficient, the RT_2DFFT implementation must be synchronous and data parallel with the whole PE array either working on a single problem instance or working on multiple problems simultaneously. For data-parallel, synchronous problems, the programmer must consider the amount and regularity of interprocessor communications and also ensure that the algorithm is implemented in a truly synchronous manner.

The parallel algorithm/parallel architecture combination for an FFT on the MasPar massively parallel SIMD architecture leaves few options when implementing this algorithm for the two latency cases—both implementations must be synchronous, tightly coupled, and data parallel. Actually, we have been able to meet all requirements with a single implementation for the MasPar that is capable of handling all possible mappings of FFTs to the architecture, regardless of whether one or more two-dimensional data arrays are processed simultaneously. For the *latency-equals-period* case, all processors must be assigned the task to work on a single data array. On the other hand, when greater latencies are permitted, multiple input arrays can be processed simultaneously—providing more workload for an individual processor and reducing the amount of interprocessor communications. Regardless of the RT_2DFFT benchmark timing requirement, efficient algorithms for SIMD architectures will be synchronous.

IMPLEMENTING TWO-DIMENSIONAL FFTS

We have implemented the RT_2DFFT benchmark using supplied library routines from MasPar to perform the FFT and to manipulate data throughout the processors. We have extensively used library software in the RT_2DFFT implementation because MasPar has attempted to optimize this software for their MPP architecture. We have examined two versions of two-dimensional FFT library software for the RT_2DFFT implementation:

1. The MasPar image processing library (MPIPL) (MasPar Computer Corp., 1992),

2. The MasPar mathematics library (MPML) (MasPar Computer Corp., 1992).

The image processing library has all data matrices arrayed in a form called a *two-dimensional hierarchical mapping*, where the data is arranged onto the processors as if both the data and processors are in a two-dimensional array. Each processor gets a localized rectangular sub-array of data. As a result, this FFT algorithm can only be used for a single matrix to PE array mapping. On the other hand, the MPML FFT routines have a more complicated mapping of data to the PE array; however, this FFT function is more extensible and permits more options.

Previous research on MIMD architectures (Brown, 1994) has shown that when there is adequate memory, the most efficient parallel two-dimensional FFT algorithm occurs when all data in a row is placed on a single processor and the two-dimensional FFT algorithm is implemented by:

1. performing a one-dimensional FFT without any communications,

2. performing a matrix transpose or corner-turn,

3. performing a second one-dimensional FFT, again without any communications.

With the MPIPL FFT functions, this algorithm scenario was impossible.

Benchmarks from MasPar confirmed that the combination of performing a pair of one-dimensional FFTs separated by a *corner-turn* or matrix transpose is the most efficient two-dimensional FFT implementation (Pickard, 1995). Due to its flexibility and potential for better performance, we have selected an FFT routine from the mathematics library. The MPML FFT routines are sufficiently extensible that we have been able to develop a single `RT_2DFFT` implementation that can handle both:

Case 1: A single input array mapped to all processors,

Case 2: Multiple input arrays mapped to the processors and processed simultaneously.

These cases correspond to the two latency scenarios in the `RT_2DFFT` benchmark specification (see section 2). By using the mathematics library FFT routines and other routines that efficiently rearrange data on the PEs, a single `RT_2DFFT` implementation can process both latency requirements of the `RT_2DFFT` benchmark.

When the entire PE array collaborates on processing a single input matrix, then the `RT_2DFFT` benchmark specifies that we select the smallest sized machine where the latency to process an input array is equal to the period. When we examine more than one input array simultaneously, the latency will be greater than the period. In fact, the only reason to examine the *latency-greater-than-period* case, is to be able to perform the processing more efficiently, and consequently, reduce the machine size.

MPML FFT IMPLEMENTATION OPTIONS

To research both latency requirements, we have included software in the `RT_2DFFT` implementation that examines all possible numbers of input arrays that can be processed simultaneously given the PE array size and memory constraints of available MasPar MPPs. The `RT_2DFFT` implementation assumes that all input array sizes and numbers of input arrays are powers of two, so that there are always perfect mappings from the input data to the PE array size. We have examined several possible mappings of input data to the processing elements for the FFT library routines. Our discussion will concentrate on three specific mappings:

1. A single input array spread over the entire PE array,

2. A single input array row per PE,

3. Multiple input array rows per PE, or multiple *tiers* of data.

The three mappings are graphically illustrated in Figure 3. In some instances, to utilize the available PE memory, it may be required to replicate the input array rows on an individual processor. This will be performed by generating multiple tiers where a single row is mapped per processor. Each tier may have multiple input arrays, and each tier of arrays is processed iteratively.

For the MPML FFT routines, the PE array is considered a one-dimensional linear array rather than a two-dimensional mesh. A single function call performs the parallel FFTs for any number of tiers, regardless of the mappings, as long the data is loaded into memory in the proper locations on the appropriate processors.

The first mapping is required for the *latency-equals-period* case when the number of processors is greater than the number of rows. For those instances where the number of processors equals the number of rows in a single matrix, then the second mapping—with $P = n$—would be used for the latency-equals-period case. In section 6, we discuss the limitations that the PE array size and the amount of memory have on these data/processor mappings.

For the *latency-greater-than-period* case, we can exploit the additional available parallelism and take full advantage of the available memory on the PE array and perform multiple FFTs concurrently. As we process multiple input arrays concurrently, there are three situations of interest. The first situation occurs when even with multiple input arrays, individual data rows must be spread over multiple processors; the second situation occurs when there are adequate data that entire rows are assigned to an individual processor; and the third situation occurs when there are sufficient data that multiple input array rows are assigned to individual PEs—arranged in multiple tiers. The first of these situations would be an extension of the first mapping in Figure 3—with multiple input arrays spread across the PE array and requiring multiple processors per row. When there is at least an entire input array row on a processor, communications when performing the one-dimensional FFTs are minimized and performance should be maximized. We illustrate this point in section 7 with empirical data.

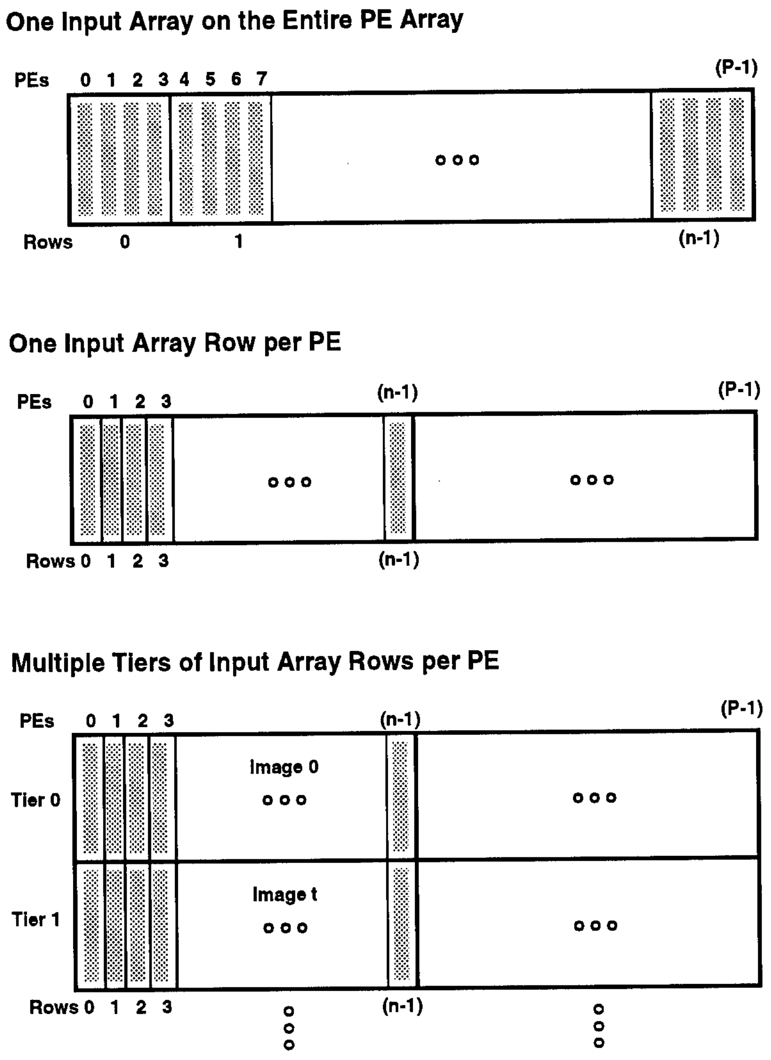


Figure 3. Mapping One or More Input Arrays onto the PE Array

MAPPING INPUT DATA TO THE PE ARRAY

We are assuming that the input data are available in the source buffer in raster-scan format, that is, in row-major format. We think of a two-dimensional input array as being in a rectangular form; however, our data is actually one long stream that will have *end-of-row* breaks imposed simply by the matrix width. Figure 4 illustrates a single input array and how the two-dimensional nature of

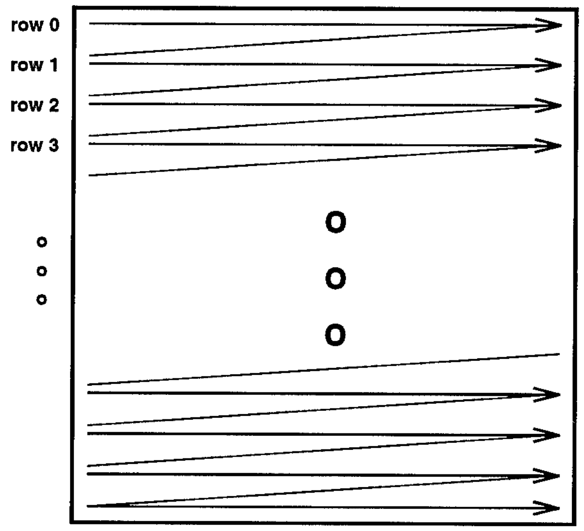


Figure 4. Conceptual Two-Dimensional Input Array

the data is actually a single data stream. Input data is stored in a single source buffer (IORAM), with row delimiters calculated from matrix size. When we are processing more than one input array at a time, they are simply arranged in consecutive order, and data location information is calculated as a function of input array size and identifier. Figure 5 illustrates multiple input arrays and the manner with which they are handled as a single data stream.

The library FFT routines specify that when more than one processor is required per input array row, the data will be mapped to the PE array in a manner referred to as *cut-and-stack*. In a cut-and-stack data/processor mapping, data within an individual row are assigned to the processors in a *round robin* manner with adjacent matrix elements being assigned to adjacent processors, then wrapped back to the first processor assigned to the matrix row. Cut-and-stack for multiple processors assigned to a single data row is illustrated graphically in Figure 6. The *cut-dimension* equals the number of processors assigned to a single row. Assigning an entire input array row to a processor is a degenerate form of cut-and-stack, where the cut-dimension is equal to one. Cut-and-stack for a single processor assigned to a single input array row is illustrated graphically in Figure 7. This figure also depicts the manner in which multiple input arrays are mapped to processors.

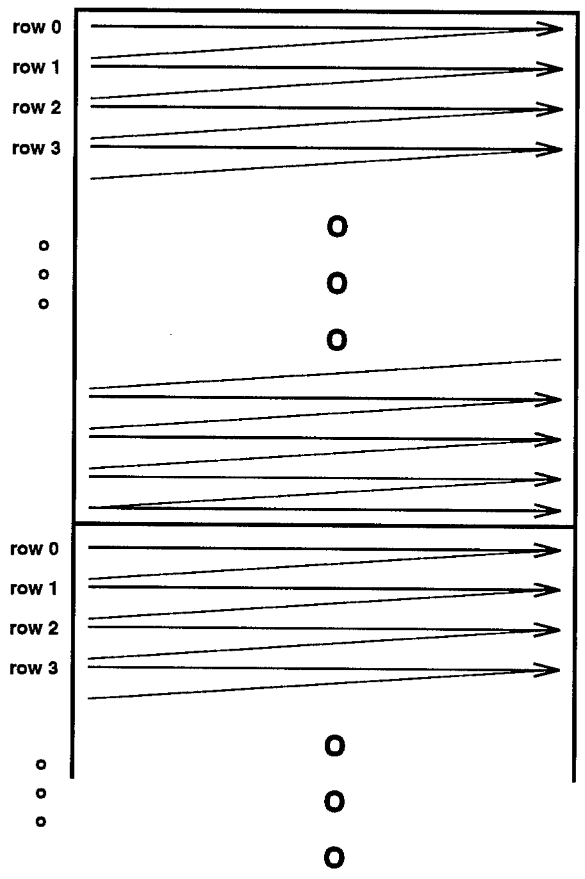


Figure 5. Multiple Two-Dimensional Input Arrays

The cut-and-stack data mapping for multiple processors per data row is, unfortunately, the worst mapping to implement directly when moving data originally in row-major order to the PE array. Only single matrix elements can be read at a time, albeit in parallel, to the PE array—a loop must be used to repeated read single values in parallel until all data is transferred to each processor in the PE array. The parallel data reads from IORAM use the router-based network and communications on this media incur communications overhead that is a linear combination of a fixed start-up latency and a message transfer cost proportional to message size. Consequently, multiple parallel data reads from IORAM incur multiple instances of fixed communications latency, inflating the overall communications costs to move source data from the IORAM to the PE array and to move the processed data from the PE array back to the IORAM. When there are

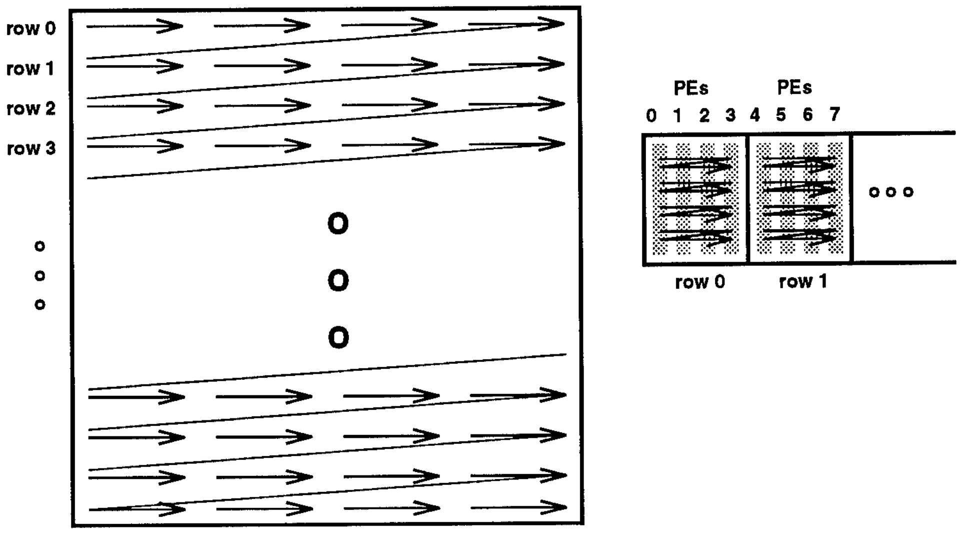


Figure 6. Cut-and-Stack for Multiple Processors per Data Row

adequate data that an entire row is mapped to a single processor, the degenerate cut-and-stack permits the data to be read in a single parallel operation.

To minimize communication costs, we developed a two step process to read and write data in parallel from and to IORAM. This technique reads a section of an input array row to a processor and then redistributes the row data to the processors in the appropriate cut-and-stack format. We save communications time by limiting the number of parallel input data read/writes, and we replace these operations with a call to a MasPar library routine that *shifts* the data to the proper storage positions on the correct processors. Figure 8 illustrates the *read/shift* procedure for a parallel read operation. This technique minimizes the number of read/write messages involving the IORAM and performs the data distribution in a manner that has been optimized for the MasPar architecture. Details on the implementation of the shift functions are provided in appendix B.

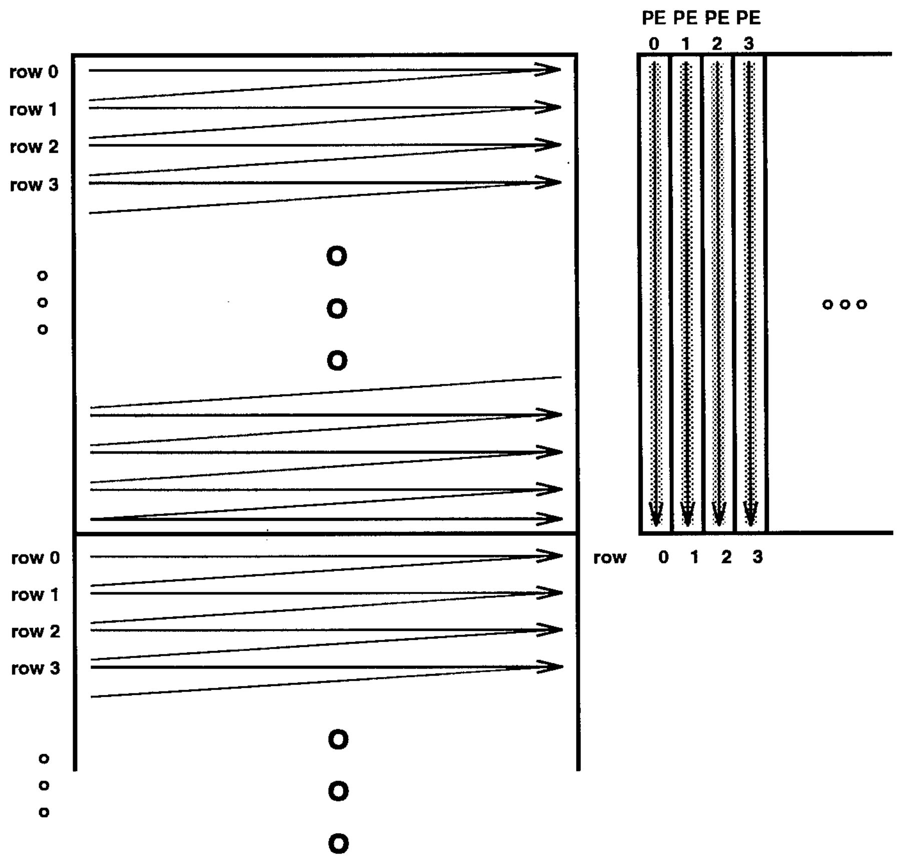


Figure 7. Cut-and-Stack for a Single Processor per Data Row

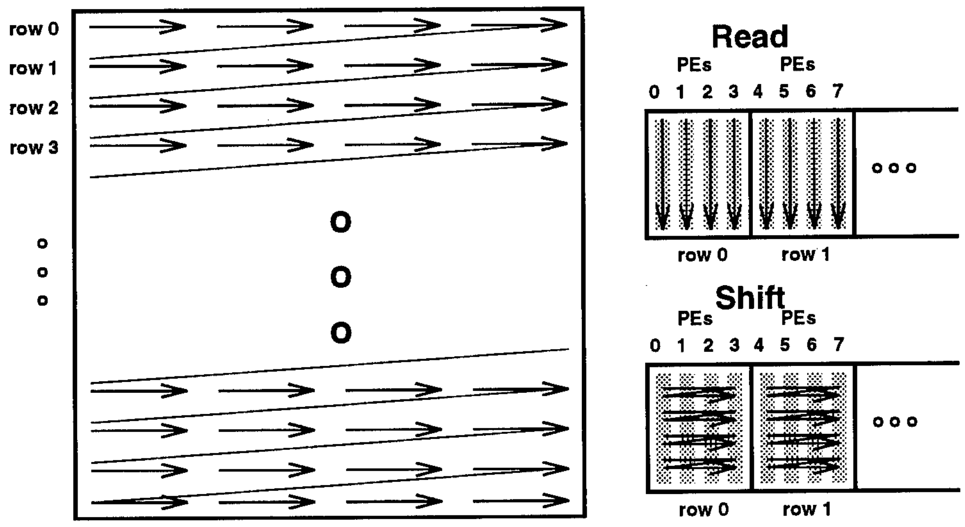


Figure 8. Read/Shift Parallel I/O

SECTION 6

MASPAR MPP ARCHITECTURE CONSTRAINTS

In this section, we examine performance constraints imposed by the MasPar MPP architecture on the RT_2DFFT benchmark implementation and on real-time performance in general. Performance of our RT_2DFFT implementation is highly dependent upon the MasPar MPP processor type and the available hardware resources: in particular, the amount of memory and the available bandwidth in the sequential I/O subsystem bus. The quantity of available memory in each of three separate memory subsystems affects performance—in particular, the amount of PE memory, the amount of IORAM, and the amount of ACU memory. Constraints imposed by each memory quantity are discussed below for both maximum hardware configurations and the actual configurations used to test the RT_2DFFT implementation. In addition, we discuss the real-time processing constraints on possible input array sizes imposed by the available bandwidth in the sequential I/O subsystem bus.

PROCESSOR TYPE

The processor type (MP-1 or MP-2), number of processors, and the amount of memory in the PE array directly affect the performance of the MasPar computational engine and the performance of our RT_2DFFT benchmark implementation. For a given processor type, the number of PEs determines the amount of computational power (in Mflop/s) that the machine can deliver. Those capabilities were summarized in Table 2. Theoretically, an MP-2 can perform five times the number of floating point operations per unit time as the MP-1. As a result, a smaller MP-2 may be applicable for a RT_2DFFT benchmark requirement than an MP-1. Also an MP-2 may be able to process larger input arrays in the specified input array inter-arrival time than an MP-1. Empirical results, presented in section 7, support these conclusions.

MEMORY

Regardless of the processing period specified in the RT_2DFFT benchmark, there are some theoretical constraints imposed on the size of input arrays that can be processed on a MasPar MPP due to maximum amounts of available memory.

The analysis in this section assumes that an entire input array must be loaded into PE array memory at the same time to perform the RT_2DFFT benchmark. Any *out-of-memory* technique would not be feasible for real-time applications. In particular, the amount of PE array memory and the amount of IORAM memory affect input array size and number of arrays that can be processed concurrently. The amount of ACU memory affects the duration of the benchmark runs.

PE Array Memory

The amount of PE array memory is a function of the PE array size and the amount of memory per processor, denoted by M. Since there are P PEs, the total memory in the PE array is $(P \cdot M)$ bytes. There must be enough memory not only to store the input array, but also to store the auxiliary plural variables required for the calculation. Assuming that we have single-precision complex input arrays, each array element is represented by eight bytes, and an $n \times n$ input array requires $(8 \cdot n^2)$ bytes. In addition, at least double this amount of memory is required for auxiliary work space. Thus we need $(24 \cdot n^2) < (P \cdot M)$. This relationship can be used to determine the largest input arrays that can be processed as a function of the PE array size and the amount of memory per processor.

Table 4 presents the largest input array that can be processed as a function of M and P. Both the NPAC MP-1 and the MasPar Corporation MP-2 used for benchmark trials had 64 Kbytes of available memory on each PE—the corresponding row in Table 4 is highlighted by the arrow. In particular, this table shows that due to memory constraints, our RT_2DFFT implementation could examine no input array of size greater than 4K $\times$ 4K. To examine 8K $\times$ 8K input arrays with our RT_2DFFT benchmark implementation, a MasPar MPP with more memory per processor would be required. It is important to note that due to memory constraints, no single-precision complex input arrays larger than 8K $\times$ 8K can be processed on either the MasPar MP-1 or MP-2.

IORAM Memory

The next memory constraint affecting performance is the amount of available IORAM, the serial memory in the I/O subsystem used as the input array source and output array sink. The IORAM must be sufficiently large that it can support both data source and data sink functions. Thus the IORAM must have at least $(2 \cdot 8 \cdot n^2)$ bytes of memory—for n = 2K, this corresponds to 64 Mbytes of storage. Four times this amount, or 256 Mbytes, are required when n = 4K. Double buffering requirements for actual processing would double these amounts. MasPar

Table 4. Maximum Input Array Size as a Function of MasPar MPP Configuration

	P				
M	1K	2K	4K	8K	16K
16K	512	1K	1K	2K	2K
$\rightarrow$ 64K	1K	2K	2K	4K	4K
256K	2K	4K	4K	8K	8K

MPPs can be purchased with up to 1024 megabytes of IORAM (a gigabyte), thus conceptually IORAM limitations would limit the potential maximum single-precision complex input array size to 4K $\times$ 4K. Actual IORAM constraints on available machines constrained our `RT_2DFFT` implementation to 2K $\times$ 2K input arrays. This was achievable only by carefully managing the IORAM memory and *re-using* the same buffer space in IORAM for both the input source and the output sink and saving the input array in PE memory.

This reuse is achieved by first copying the source input array into PE array memory before any timing information is collected. After we begin the benchmark process, the initial time stamp is calculated, the input array is read in from the data source, and the FFT is calculated. However, instead of writing the processed array to the shared buffer, the stored input array is written out to the buffer. Subsequent iterations of the benchmark repeat this cycle. After the `RT_2DFFT` benchmark has completed and all timing information has been gathered, the last processed array is sent to the data sink buffer for verification processing.

ACU Memory

The last memory-based implementation constraint we encountered is a result of the amount of serial memory available in the ACU. This problem is a side-effect of the `RT_2DFFT` benchmark; in particular, timing information is stored on the ACU when running the `RT_2DFFT` implementation to remove the impact of accessing the UFE OS during a benchmarking trial. However, due to constrained amounts of ACU memory on the available MasPar machines, some of the longer benchmark runs—15 minutes in length as specified by the RT_2DFFT benchmark—were impossible. This limitation is only a factor when the time to process an input array is significantly less than a second.

I/O SUBSYSTEM BANDWIDTH

In addition to the RT_2DFFT implementation constraints imposed by finite memory resources, another constraint on maximum data throughput exists: the finite bandwidth of the bus within the I/O subsystem. Even though this bus has the massive bandwidth of 230 Mbytes/s (megabytes-per-second), it can be a sequential bottleneck. Due to bandwidth limitations, it is impossible to simultaneously load 4K $\times$ 4K unprocessed input arrays from an external source and send the processed data to the frame buffer for display if the data arrives at a rate of only one array-per-second. The bandwidth required to transfer 4K $\times$ 4K single-precision complex input arrays into and out of IORAM at a rate of only one array-per-second would be 256 megabytes-per-second $(2 \cdot (8\text{bytes} \cdot (4K \cdot 4K)))$.

During our experimentation with the MasPar RT_2DFFT implementation, we never encountered this problem because the RT_2DFFT benchmark explicitly does not require that external I/O be considered at this step. Theoretically, we could configure the IORAM with sufficient memory to support 4K $\times$ 4K input arrays; however, unless the bus bandwidth is significantly increased it would not be possible to move data to and from IORAM. Consequently, the finite bandwidth sequential I/O channel bus is an inherent limitation that will effect the extensibility of the present overall MasPar architecture to real-time processing of large matrices.

CONCLUSION

In this section, we described various constraints attributable to the MasPar MPP architecture for two-dimensional applications with single precision complex data array elements. Constraints on maximum PE memory impose a potential limitation on input array sizes to 8K $\times$ 8K rows and columns. Furthermore, maximum IORAM memory constraints reduce the maximum input array size to 4K $\times$ 4K rows and columns, and maximum I/O subsystem bandwidth constraints further reduce the maximum input array size to only 2K $\times$ 2K rows and columns. These constraint numbers assume that the problem output is as large as the problem input array, as specified in the RT_2DFFT benchmark. The maximum input array size of 2K $\times$ 2K falls significantly short of the 16K $\times$ 16K maximum input array size stated in the RT_2DFFT benchmark specification. However, in many applications, entire two-dimensional arrays are read in and processed, but only minimal information, e.g., detections, are output. For applications that do not produce output streams with as much data as the input stream, the size of the problem that could be considered with current systems would effectively double.

SECTION 7

MASPAR PERFORMANCE RESULTS FOR THE RT_2DFFT BENCHMARK

In this section, we describe the real-time performance of our RT_2DFFT benchmark implementation. We have exercised the MasPar RT_2DFFT implementation for a variety of input array sizes, machine sizes, and the latency cases—all configuration values are specified in the RT_2DFFT benchmark. The hardware configurations for both the NPAC MP-1 machine and the MasPar Corporation MP-2 machine on which we collected the data are described above in section 3.

The RT_2DFFT benchmark specification requires that basic clock functions be examined to form a baseline with which to measure real-time performance. Properties of the clock and the methods employed to time our performance runs are described in the first subsection. After the timing subsection, there are five subsections that evaluate the empirical data collected when exercising our implementation. We first compare two techniques for moving the input data from the data source to PE array memory and the processed data from the processor memory to the data sink. We present empirical data to illustrate the tradeoffs between performing I/O directly with the IORAM and by performing the I/O in a more convenient manner but shifting the data to the proper PE array memory locations. In the next subsection, we examine empirical data collected during extended RT_2DFFT implementation tests with durations as long as 15 minutes. Empirical performance data collected during these extended runs permit the assessment of any effect that the UFE OS may have on the real-time performance of our implementation. We have found that real-time performance is predictable with the most variance in empirical timing data due to the side effects imposed by calling the clock.

Because of the predictable near absence of variation observed when running the MasPar RT_2DFFT benchmark implementation, the last three subsections focus on a parametric analysis based on a set of shorter runs, typically consisting of only ten iterations. In these subsections, we only present a summary of the data; an extensive listing of the empirical data is tabulated in appendices C and D for the MP-1 and MP-2 respectively. We first describe the empirical results for processing a single input array at a time—relevant to the latency-equals-period case in the RT_2DFFT benchmark specification. We next describe empirical results for processed multiple input arrays concurrently—relevant to the latency-greater-than-period case. In the final subsection, we examine scalability of a real-time, embedded system. We first identify the smallest machine—for both MP-1 and MP-2—that is required to meet the RT_2DFFT benchmark specification of a one-second input array inter-arrival time or maximum output period for various input array sizes. Then we examine sustained processor utilization as a function of input array size and machine size.

TIMING MEASUREMENTS ON THE MASPAR MPP

Access to a clock in our RT_2DFFT benchmark implementation used MpTimer routines that measure elapsed time using the clock capabilities in the UFE. The RT_2DFFT benchmark specifies that *wall clock* time be used in the implementation. The MasPar clock function meets this requirement—it continues to run even when the implementation, running on the ACU and PE array, is swapped out or otherwise interrupted.

We were unable to use a high resolution timer on the ACU because of unspecified implementation problems with the function calls. It would have been desirable to have had a timer located on the ACU and PE array to eliminate any interaction with the UFE. It would also have been desirable to have a higher resolution clock than was available for use.

To test timing variability, we ran the repeated clock benchmark as described in (Brown, 1994). The clock benchmark repeatedly calls the elapsed timing routine in a tight loop and stores the elapsed times between calls. Differences between successive times, called *deltas*, can be analyzed during post processing to measure clock variability. The measured deltas can be used to examine the clock resolution of the timing functions. When the elapsed times between clock calls is less than the clock resolution, the data will be reported as either zeros or the minimum clock resolution. Clock resolution depends on the UFE type and its OS. For the MP-1, we were limited to a resolution of about 4 milliseconds; on the MP-2 the resolution was about a 1 millisecond. The actual time to perform the function call to obtain elapsed time varied, often greatly. We suspect that a clock call is highly dependent on UFE activity.

Effects due to the UFE can be inferred from the number of clock calls made within the clock resolution and from the size of measured deltas. For example, on the MP-1 UFE, we found the number of clock calls that could be completed within the clock resolution—about 4 milliseconds—varied from 18 to 60 calls. The distribution of the calls was trimodal with peaks at about 20, 40, and 60 calls. The worst case, when only 20 calls were completed in 4 milliseconds, indicates that about 200 microseconds are needed for the clock call. Although we did not run the clock benchmark on the UFE of the MP-2, we did run it on a DEC Alpha Workstation with the same OS. We found that the number of calls within the one-millisecond clock resolution varied between 243 and 287 with distribution peaks at 274 and 286.

Besides the variability of the clock call time, there is also variability indicated by deltas being larger than the resolution. These large deltas are due to process activity on the UFE—the clock call must wait until the OS can process it. OS activity can be a result of other users being processed by the OS; however, there are also many system daemons that the OS must service periodically. In one clock benchmark on the MP-1, we noticed a delta of 100 milliseconds. This large delta indicates the difficulty in measuring real-time performance using a non-real-time OS. Timing using the test bench should not induce side-effects that are more significant than the predictability we are attempting to measure. We attempted to minimize side-effects by being sole user on the MasPar UFEs as we ran our RT_2DFFT implementation.

The differences that we have noted between machine types are only one of scale. The same problem exists on either UFE machine type—OS activity—ignoring the interference of other user processes.

The RT_2DFFT benchmark implementation processing loop consists of the following operations:

1. Read data from the IORAM to the PE array,
2. Compute 1D FFTs on the rows,
3. Perform a corner turn on every data array,
4. Compute 1D FFTs on the columns,
5. Write the data from the PE array to the IORAM.

We implemented two types of clock insertions; they are illustrated in the two diagrams in Figure 9. Each diagram shows the processing operations in the loop. The top diagram shows that *block timing* has clock insertions between each

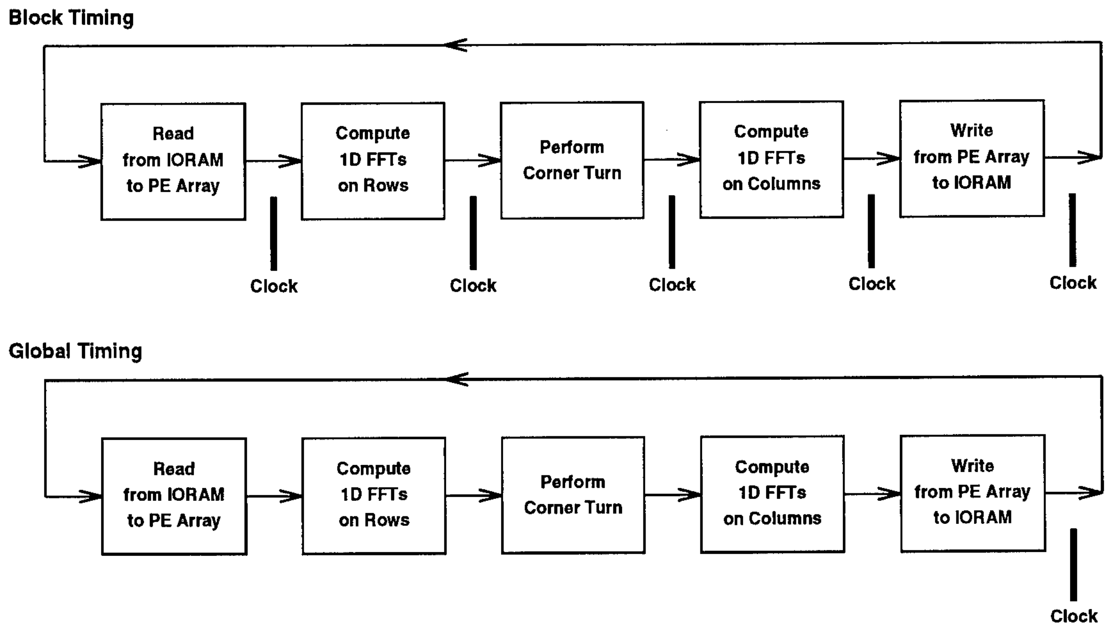

Block Timing
Read from IORAM to PE Array
Compute 1D FFTs on Rows
Perform Corner Turn
Compute 1D FFTs on Columns
Write from PE Array to IORAM
Clock
Clock
Clock
Clock
Clock
Global Timing
Read from IORAM to PE Array
Compute 1D FFTs on Rows
Perform Corner Turn
Compute 1D FFTs on Columns
Write from PE Array to IORAM
Clock

Figure 9. Timing Insertion Options

operation to time each separately; this requires five separate calls to the system clock. The bottom diagram in Figure 9 shows that *global timing* has only one clock insertion in the complete processing chain, only one call to the system clock is made during each loop iteration. The clock insertion method is specified at run time. Except for the time required to perform the additional four clock calls, we expect that the sum of times for the individual operations in block timing should be similar to global timing. Timing variability for block timing will be greater than timing variability for global timing because of each of the multiple timing calls contributes to an aggregate variability.

Since a single clock on the UFE was used, there was no requirement for synchronizing clocks at the processors or at the data source and data sink. A single clock insertion was placed at each of the locations within the code as indicated in Figure 9.

PARALLEL INPUT COMPARISON

We examined two parallel I/O implementations: one that reads input array data directly to processors in the required *cut-and-stack* data mapping, and another that reads the input array data in a more efficient manner by portions of rows but then requires data communications to shift the data into the required cut-and-stack data mapping. Empirical data shows that, in all situations of interest, the *read/shift* operation for parallel data array I/O from/to the IORAM is more efficient than cut-and-stack I/O operations. We present data in Table 5 to illustrate the improved performance of the read/shift operation compared to the cut-and-stack for multiple processors per data array row. Table 5 presents data for comparing the time to perform:

1. One parallel input array read from IORAM,

2. One parallel output array write to IORAM,

3. One loop combining the I/O and the two-dimensional FFT.

The results are for 512 $\times$ 512 input arrays and for 1K to 16K processors. The timing data are for distributing a single input array over all processors. The data were collected in short duration benchmarking runs performed on the MP-1. The system configuration for this machine is presented in section 3.

In all PE array size combinations reported in Table 5, read/shift is better than cut-and-stack. The results are representative of all input array sizes examined in this analysis; however, other experiments did show that cut-and-stack is superior for small input arrays (256 $\times$ 256) on 8K and 16K processor machines. In these cases, the time to process the single input array is significantly less than the time required to perform the operation as specified by the `RT_2DFFT` benchmark, so a smaller machine would be used.

We conclude that for parallel I/O, it is better to read from the IORAM in the fastest way possible using parallel strides, and then to add the interprocessor communication step to rearrange the data to the correct data mapping. This shift operation is similar to the corner turn that we use between individual one-dimensional FFTs as we perform a two-dimensional FFT. Appendix B contains a more detailed discussion of how these rearrangements are implemented.

Table 5. **Comparison of Row/Shift versus Cut-and-Stack Data Array I/O—512 × 512 Array**

P	I/O Algorithm	Times in milliseconds Read	Write	Loop
1K	read/shift	312	258	1094
	cut-and-stack	469	426	1414
2K	read/shift	163	137	593
	cut-and-stack	243	226	755
4K	read/shift	78	67	302
	cut-and-stack	125	106	382
8K	read/shift	27	25	134
	cut-and-stack	63	55	200
16K	read/shift	19	20	86
	cut-and-stack	20	20	82

EXTENDED-DURATION TESTS

One main requirement in the RT_2DFFT benchmark specification is that the implementation should be run for 15 minutes. Given the requirement for a one-second maximum output period, we are examining at least 900 iterations that should give an adequately large sample to fully understand real-time performance predictability. Besides confirming that period requirements can be met, the extended-run time also allows us to assess what effect, if any, the operating system has on the real-time performance. Such an effect would manifest itself as some processing times being significantly larger than the rest. We use maximum latency and maximum output period as the metrics to describe performance in a benchmark run to ensure predictable real-time performance. During a run, timing information is stored on the MasPar ACU using either block or global timing as described above. The initial iteration of the processing loop is ignored. All analysis is performed in post processing.

To ensure that there are no drastic timing effects due to our benchmark implementation being interrupted and *swapped-out*, we only collected timing data when we were the sole process running on the ACU and PE array. The extended runs that we show are all based on processing a single input array at a time, so they are relevant to the latency-equals-period case. Similar extended run results for multiple input arrays have been analyzed, but they are not presented here, because like the single input array cases, they did not show anything that demonstrates a significant disruption to real-time performance.

We first present the empirical data for a 15 minute run on the MP-1 in Figure 10. This figure contains three graphs:

1. A time plot for the 15 minute run,

2. A histogram for block timing,

3. A histogram for global timing.

The top graph in Figure 10 shows a typical 15-minute RT_2DFFT implementation run on an MP-1 configured as a 1K machine. A single 512 $\times$ 512 input array was processed by the entire PE array. The data in this graph were generated using block timing and the graph displays the accumulated time for each operation in the processing loop. Thus the top curve (labeled "IORAM Write") marks the total time to process a single loop iteration. The x-coordinate indicates the time in seconds when the processing of the input array started. This graph shows the relative time spent between I/O and FFT processing; slightly more than half of the time is spent moving data from the data source to the PE array and from the PE array to the data sink. Notice that the total time required is consistently around 1100 milliseconds; in particular, this run shows that a 1K MP-1 cannot meet the target maximum output period specification of one-second.

The thickness of the curves in the top graph of Figure 10, which is a plotting artifact due to the size and closeness of the characters used to represent the data points, makes it difficult to examine variability during the 15 minute run. The bottom portion of Figure 10 shows two histograms that are better suited to examine processing variability. On the left we present a histogram derived from block timing, in particular, for the times corresponding to the "IORAM Write"

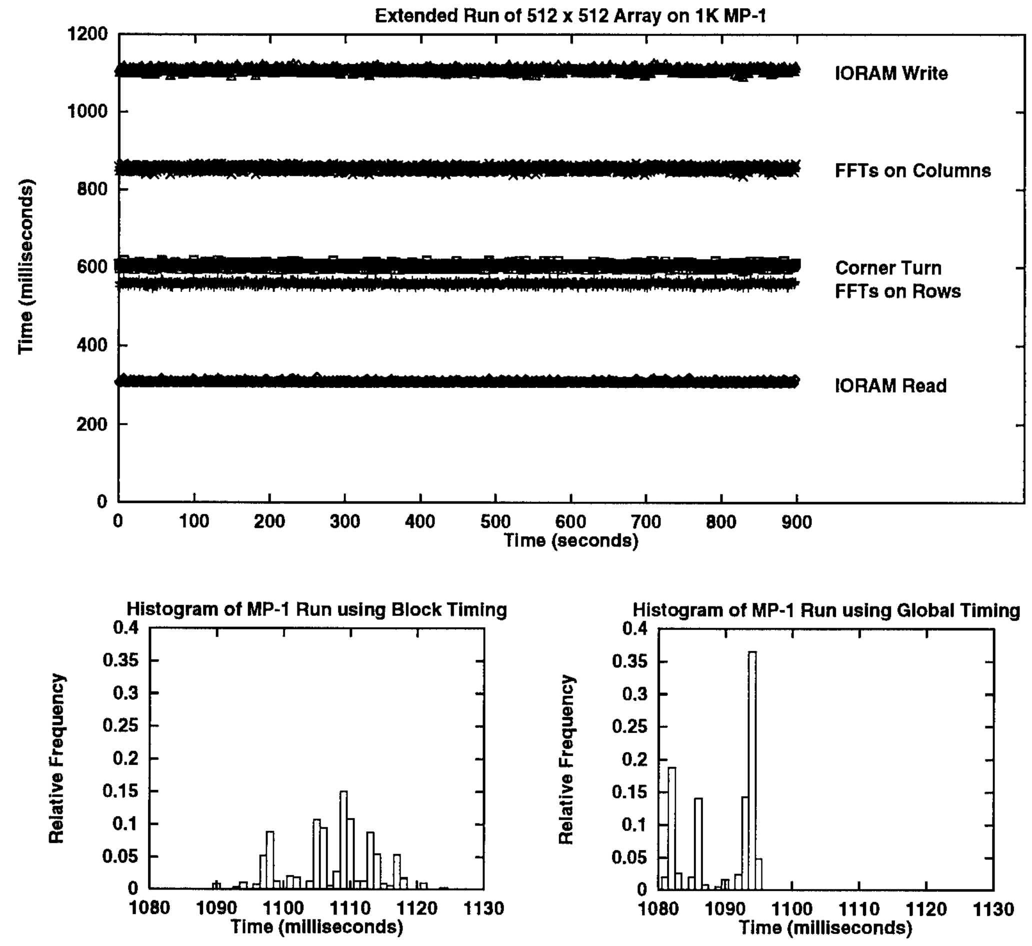


Figure 10. Processing a Single 512 × 512 Input Array on a 1K MasPar MP-1

curve. There is approximately a 40 millisecond range in the histogram. This range is about 4% of the total time for processing the entire loop (about 1100 milliseconds).

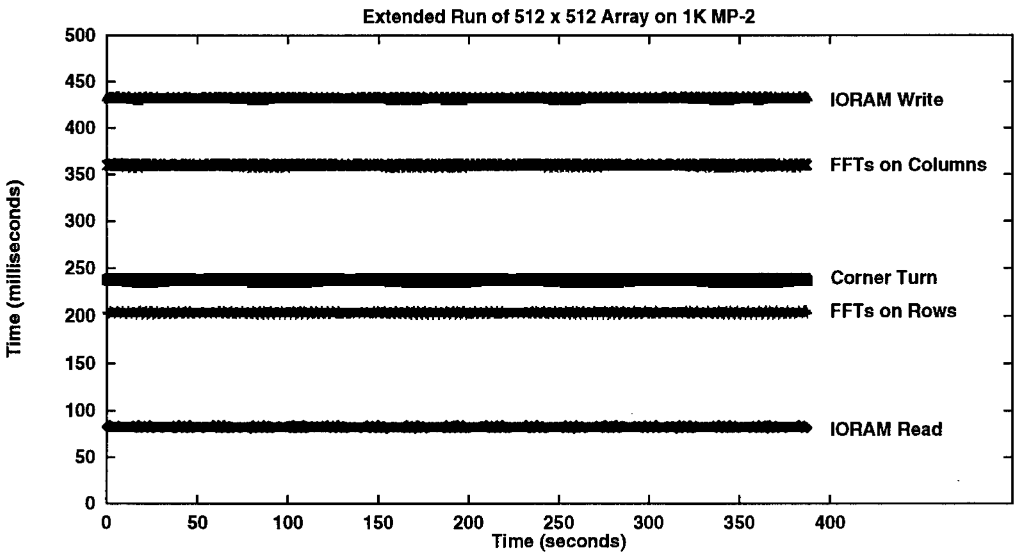
Since the times for the left histogram in Figure 10 are based on summing the timings for the five operations—requiring five separate calls to the clock—we were concerned that the variability was influenced by the additional clock calls. We reran the 15 minute test using global timing and present a histogram for this timing option on the right in the figure. The range has dropped to only about 15 milliseconds, which results in a more peaked histogram. The arithmetic mean time has shifted lower by about 20 milliseconds. These changes can be explained by the reduced number of clock calls. The results are consistent with timing behavior observed in the clock benchmarks run on the MP-1 UFE (discussed above).

In Figure 11, we present empirical performance graphs for the MP-2 machine. The presentation is similar to Figure 10, except that we include only a single histogram for block timing data. We again process a single 512 $\times$ 512 input array with the machine configured with a 1K PE array. The top graph in Figure 10 shows the accumulated block times for a single loop to process the input array. The run duration was only six minutes, due to ACU memory limitations for storing the timing arrays. The timing curves in Figure 11 illustrate less variability than those in Figure 10. We present a histogram—based on block timing—in the bottom portion of Figure 11 to examine this variability. The range of data in the histogram is only six milliseconds, which is about 1.5% of the total time. The reduced variability for the MP-2 run is most likely due to the clock, which has both better resolution and a faster call time for the Alpha workstation-based UFE.

Although not presented here, we have examined graphs similar to those in Figures 10 and 11 for both machine types and various input array sizes, various machine sizes, and various numbers of simultaneously processed input arrays. Almost all of the extended runs for both the MP-1 and MP-2 showed the same minimum amount of variability that we have observed in these figures. The few exceptions were most probably due to the presence of other users on the UFE or to operator interference.

LATENCY-EQUALS-PERIOD CASE

This section describes empirical results for processing each input array as it arrives—corresponding to the case where the latency must equal the period. Because of the extremely small amount of run-time variability that we observed above for the extended duration tests, this in-depth parametric performance analysis has been based on data obtained in shorter duration tests of the MasPar `RT_2DFFT` implementation. We collected statistics for runs of only ten iterations of the processing loop after ignoring the initial iteration. All runs have been duplicated for both block and global timing. Note that the total time for the run will vary from the sum of the pieces, since they are based on separate timing insertions and separate runs. A complete listing of the empirical performance data is presented in appendix C for the MP-1 and in appendix D for the MP-2.



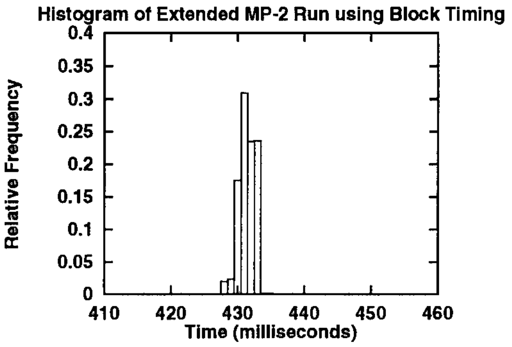


Figure 11. Processing a Single 512 × 512 Input Array on a 1K MasPar MP-2

In Table 6, we present the maximum processing period for a single input array using global timing for all available machine sizes and for four different input array sizes, 256 × 256, 512 × 512, 1K × 1K, and 2K × 2K. As we examine real-time performance, we are interested in examining variability. The maximum processing period represents the worst-case situation, and is an estimate of the time required for the operation to be processed and maintain the predictability required of a real-time system. An "—" indicates that a timing value is not available for those parameters. The current version of the RT_2DFFT benchmark implementation cannot process a 2K × 2K input array on a 1K machine—we have not implemented a two-dimensional FFT algorithm where there are multiple rows

Table 6. Maximum Times in Milliseconds for Latency Equals Period

	MP-1					MP-2		
n	1K	2K	4K	8K	16K	1K	2K	4K
256	282	126	54	51	40	119	67	31
512	1094	593	302	134	86	427	249	131
1K	3359	2341	1411	684	313	910	964	525
2K	—	7539	5641	2969	1500	—	2202	2079

Table 7. Sustained Processing Rate in Mflops/s for Latency Equals Period

	MP-1					MP-2		
n	1K	2K	4K	8K	16K	1K	2K	4K
256	18.6	41.6	97.1	102.8	131.1	44.1	78.3	169.1
512	21.6	39.8	78.1	176.1	274.3	55.3	94.8	180.1
1K	31.2	44.8	74.3	153.3	335.0	115.2	108.8	199.7
2K	—	61.2	81.8	155.4	307.6	—	209.5	221.9

from the same input array being assigned to a single processor. We have not been concerned with this implementation scenario because there is not enough PE array memory on the available machines to run this case. The empirical timing data in Table 6 has been used to generate Table 7, in which we present the sustained processing rate in Mflops/s for the latency-equals-period case. The calculation of Mflop/s uses $10n^2 \log_2 n$ for the number of operations.

The times in Table 6 indicate how the period scales with respect to input array size and machine size. Similar scalings are apparent in Table 7 in the Mflops/s rate. The times for the MP-1 decrease and the Mflops/s increase as the machine size increases—as expected—but sometimes in a super-linear manner (consider

256×256 input arrays). This speed-up could be an artifact of the software-based technique to reconfigure the machines to simulate smaller machines. The times and Mflops/s rate for the MP-2 show an almost linear speed-up with respect to machine size, except for the 1K $\times$ 1K input array on the 1K and 2K machines. In this case, the processing time actually worsens when the machine size is increased. This is caused by the relationship between computation and communication capabilities on the MP-2. This machine has faster processors but the same communication routers and networks as the MP-1. When expanding the machine size from a 1K to a 2K PE array for a 1K $\times$ 1K input array, we spread a row of data over two PEs rather than just one. Additional interprocessor communications are required, which results in an increase in communications time that evidently is greater than the decrease in computation time due to the added parallelism.

Scaling is approximately linear with respect to the size of the input array, n^2. That is, if n doubles, we expect about a factor of four increase in the processing time. The data in Table 6 scale in this manner, except in the situation when a row is mapped to a single PE on the MP-2. For example, the time only roughly doubles when going from a 512 to a 1K input array on a 1K MP-2. The reason for this anomaly lies again in the reduced internal communications that occur when an entire input array row is mapped to a PE.

If we compare comparable parameter values for the MP-1 and MP-2 in Tables 6 or 7, there generally is a speed increase of a factor of 2–3 for an MP-2 versus an MP-1. However, MasPar advertises that there is a 5 fold increase in peak floating point processing rate for an MP-2 compared to an MP-1 (see Table 2). We are not able to obtain this increase in performance with the `RT_2DFFT` benchmark implementation, because the MP-2 communications capabilities have not been improved.

For the performance of any parallel application implementation to scale with processor improvements in a new architecture, the communications capabilities must improve as much as the processor performance. Formulas are provided in (Koester, 1995) that quantify this relationship, and they can be used to estimate performance if communications and calculations capabilities do not increase equally.

LATENCY-GREATER-THAN-PERIOD CASE

This section describes empirical results for processing multiple input arrays simultaneously—corresponding to the case where the latency can be greater than the period. As in the previous subsection, our results are based on short runs that are typically ten replications of the processing loop after ignoring the initial iteration. All runs have been duplicated for both block and global timing. A complete listing of the empirical performance data is presented in appendix C for the MP-1 and in appendix D for the MP-2.

If latency is permitted to be greater than the period, we show that it may be possible to reduce the machine size by processing multiple FFTs simultaneously. In particular, if an entire row of a two-dimensional input array is assigned to a single processor, we will show that the RT_2DFFT benchmark runs more efficiently, because communications overhead is minimized during both I/O and FFT processing. In this analysis, we are interested in total time to process the block of input arrays divided by the number of input arrays being processed simultaneously. This normalized single problem processing time must be less than or equal to the one second period specification. This notion of maximum output period will be used extensively as the metric to evaluate RT_2DFFT implementation performance for this latency case.

The RT_2DFFT benchmark implementation is designed to process any power-of-two numbers of input arrays up to the memory capacities of the machine. In fact, we have found that we only need to consider two cases: processing one input array, which we considered above for the latency-equals-period case, and processing one full tier of input arrays for the latency-greater-than-period case. A *full tier of input arrays* occurs when we process the precise number of input arrays that the aggregate number of input arrays rows equals the number of processors in the PE array. In Figure 12, we present empirical (worst-case or maximum output period) performance data that illustrates this claim. In this graph, we show three families of curves for the maximum output period for various number of input arrays when we process 512×512 input arrays on the MP-2 machine. There are individual curves for the three available machine sizes. Each of the three curves have a similar shape—maximum output period decreases as additional input arrays are processed simultaneously, then each curve levels off when there is exactly one input array row per PE. The number of input arrays required to achieve a full tier with one input array row per PE is indicated in the legend of Figure 12. Performance does not improve for additional tiers of input arrays: therefore, during the remainder

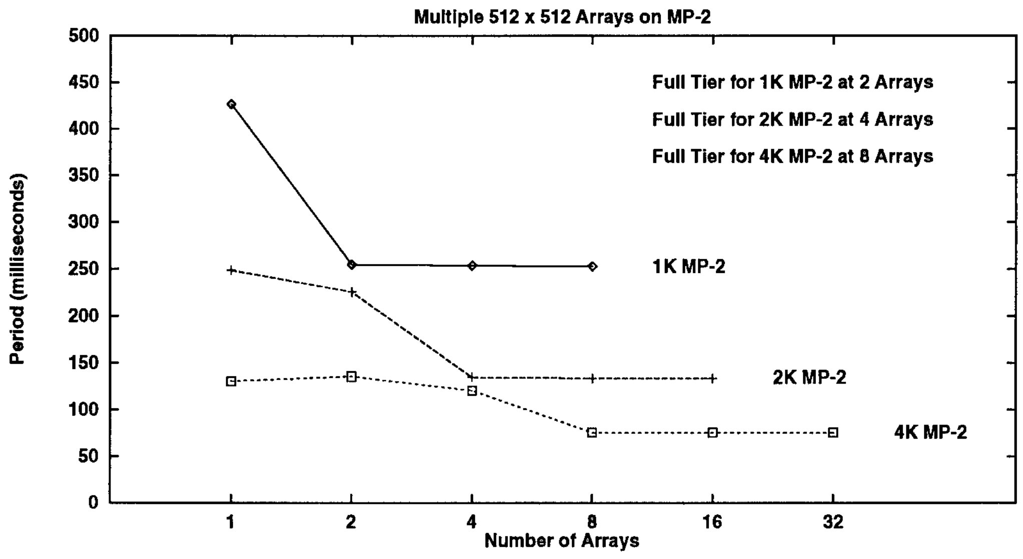


Figure 12. Period for Multiple 512 $\times$ 512 Input Arrays

of this subsection, we only consider processing a single, full tier of input arrays when discussing the performance for the latency-greater-than-period case.

In Table 8, we present the maximum run times for both latency and period for all machine and input array sizes. For each machine size and input array size, we give the number of input arrays processed (N_A) followed by the latency (Lat) and the normalized time per input array or maximum output period (Per). In general, the entries are from configurations with a full tier of input arrays, with a full input array row assigned per PE. There are two exceptions:

1. The entry marked with a † appears to be an anomaly and may be an artifact of the configuration software.
2. The entries marked with a ‡ are cases when a full tier of arrays could not be processed due to the size limit of the IORAM.

We ignore these exceptions as we analyze the data. In a manner similar to the latency-equals-period case, the empirical timing data in Table 8 have been used to generate Table 9, in which we present the sustained processing rate in Mflops/s per input array for the latency-greater-than-period case.

Table 8. Maximum Times in Milliseconds for Latency Greater than Period

n		MP-1					MP-2		
		1K	2K	4K	8K	16K	1K	2K	4K
256	N_A	4	8	1[†]	32	64	4	8	16
	Lat	793	793	54	1031	1031	219	221	251
	Per	198	99	54	32	16	55	28	16
512	N_A	2	4	8	16	16[‡]	2	4	8
	Lat	1743	1727	2180	2187	1345	510	539	608
	Per	872	432	273	137	84	255	135	76
1K	N_A	1	2	4	4[‡]	1[‡]	1	2	4
	Lat	3359	3466	4379	2828	313	910	1071	1212
	Per	3359	1733	1095	707	313	910	536	303
2K	N_A		1	1[‡]	1[‡]	1[‡]		1	2
	Lat	—	7539	5641	2969	1500	—	2202	2589
	Per	—	7539	5641	2969	1500	—	2202	1295

[†] Anomaly perhaps due to configuration software.

[‡] A full tier of input arrays could not be processed due to IORAM size limitations.

The maximum output period in Table 8 and Mflop/s per input array in Table 9 scale nearly linearly with respect to machine size; the maximum output period is halved as the machine size doubles and the Mflop/s per input array doubles with machine size. Furthermore, the maximum output period scales with input array size as we expect: a factor of four increase in time as n doubles. In particular, the effect that we observed when processing a single input array with the 1K and 2K MP-2—the maximum output period increasing as the machine size increased—does not occur when we process only full tiers of input arrays. The explanation for this is simple—as we increase machine size we also increase the number of input arrays, always mapping an entire input array row to a PE. Consequently, there never is an increase in the amount of communications.

Table 9. Sustained Processing Rate in Mflop/s for Latency Greater than Period

n		MP-1 1K	MP-1 2K	MP-1 4K	MP-1 8K	MP-1 16K	MP-2 1K	MP-2 2K	MP-2 4K
256	N_A	4	8	1[†]	32	64	4	8	16
	Mflop/s	26.4	52.9	97.1	162.7	325.5	95.8	189.8	334.2
512	N_A	2	4	8	16	16[‡]	2	4	8
	Mflop/s	27.1	54.6	86.6	172.6	280.7	92.5	175.1	310.4
1K	N_A	1	2	4	4[‡]	1[‡]	1	2	4
	Mflop/s	31.2	60.5	95.8	148.3	335.0	115.2	195.8	346.1
2K	N_A		1	1[‡]	1[‡]	1[‡]		1	2
	Mflop/s	—	61.2	81.8	155.4	307.6	—	209.5	356.4

[†] Anomaly perhaps due to configuration software.
[‡] A full tier of input arrays could not be processed due to IORAM size limitations.

We can also use the data in Table 8 and Table 9 to examine the actual performance improvement for the MP-2 over the MP-1. The empirical performance of the MP-2 shows a speed up of 3–4 over the MP-1. We see a greater increase in performance here than in the latency-equals-period case because interprocessor communications are eliminated while performing the one-dimensional FFTs—each entire row is placed on a single processor.

Finally, comparing Table 6 with Table 8 shows how processing multiple input arrays can reduce the machine size. For example, consider 512 $\times$ 512 input arrays on an MP-1—a single input array has a maximum output period of 1094 milliseconds on a 1K machine: thus a larger 2K machine would be required to meet the period equals one second input array inter-arrival specification. On the other hand, the maximum output period for two input arrays processed simultaneously is only 872 milliseconds on the 1K machine. Thus the improved efficiency of processing two input arrays simultaneously permits a smaller machine size to meet the timing specification. In the next subsection, we extend this analysis as we examine real-time scalability.

SCALABILITY OF REAL-TIME EMBEDDED SYSTEMS

The fundamental goal of the RT_2DFFT benchmark specification is to examine scalability of real-time embedded high performance computing systems. The objective of the RT_2DFFT benchmark scalability study specification, presented in appendix A, requires that we determine the minimum machine size that meets the real-time requirement of fixed maximum output period. For reasons described in section 6, we have only been able to examine a subset of the input array sizes required by the specification—256 × 256, 512 × 512, 1K × 1K, and 2K × 2K.

The RT_2DFFT benchmark specification requires two separate graphs be generated to demonstrate scalability of the implementation. We first examine the minimum machine size as a function of problem size, then we examine the sustained processing utilization percentage as a function of problem and machine size.

Minimum Machine Size

Using the data presented in Tables 6 and 8, we can determine the smallest MP-1 and MP-2 that can meet the one-second input array inter-arrival time specification for the two latency cases. We present two graphs in Figure 13 to identify smallest machine size for the two latency cases. Both graphs use the machine names as point labels and "MP-1,2" indicates that both machines achieved the specification with that machine size. The left graph shows the minimum machine size for the latency-equals-period case. Neither the MP-1 nor the MP-2 could process a 2K input array in the required time, even with the maximum configuration of a 16K PE array for the MP-1. The right graph shows the minimum machine size for the latency-greater-than-period case. Although an 8K MP-2 was not available to test the RT_2DFFT implementation, it seems likely that an 8K MP-2 could meet the specification for input arrays of size 2K by processing a full tier of four input arrays.

These plots assume that the necessary amount of IORAM is available and that the memory per PE is 64 Kbytes (recall Table 3). Note that having more PE memory available would not effect these plots, because the number of input

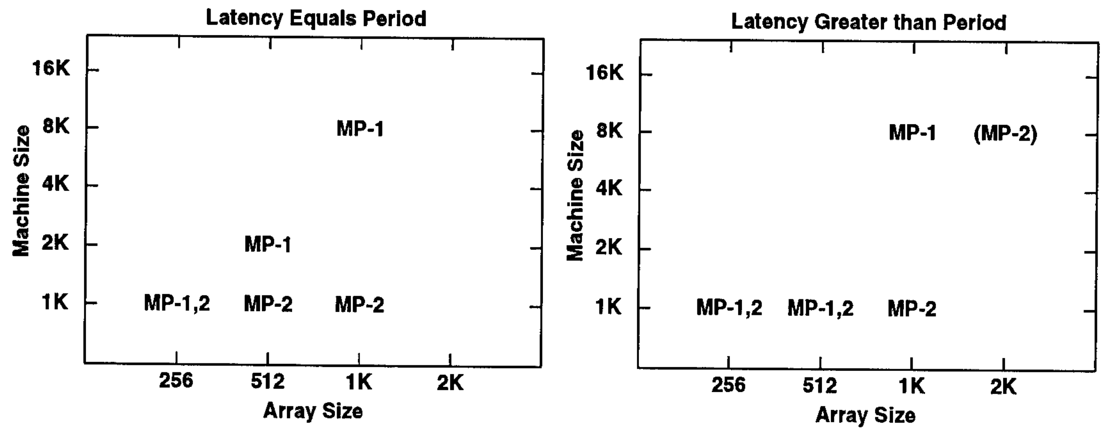


Figure 13. Smallest Machine Size for Various Input Array Sizes

arrays in a full tier is independent of the PE memory as long as there is enough memory to accommodate a whole row on a PE.

The only difference between the two plots in Figure 13 is the case of 512 $\times$ 512 input arrays on MP-1s, highlighted above in the previous subsection, and presumably the 2K $\times$ 2K input arrays. The fact that removing the latency restriction does not reduce the machine size in more cases is due to numerous factors. The primary factor is that MasPar machines are available only in discrete power-of-two sizes; so, the graphs in Figure 13 are relatively sparse. Because of the discrete nature of the number of processors, varying the specified maximum output period may cause this graph to look substantially different.

This graph does illustrate two interesting scalability issues for the growth in machine size for two-dimensional problems. First, for input array sizes 256–1K, especially with the faster MP-2, our RT_2DFFT benchmark has period less than the specified one second—often substantially less. In these cases, the minimum machine size—1K processors—will be chosen. Secondly, after the maximum period for the 1K machine is greater than the input array inter-arrival time, then the graphs in Figure 13 show an interesting fact—the increase in machine size is a factor of four for every time the number of rows is doubled. This shows some potential real-time scalability challenges when attempting to use the MasPar architecture for two-dimensional problems. As we scale the size of the problem, the MasPar MPPs will require the maximum 16K processors after doubling the input array size only twice.

Sustained Processing Utilization

To conclude this section on real-time scalability, we examine the sustained processing utilization percentage as a function of problem and machine size. Sustained processing utilization percentage is defined in appendix A as the quotient of the sustained processing rate divided by the theoretical peak processing rate. The sustained Mflops/s processing rate is defined as $10n^2 \log_2 n/(\text{maximum period})$. Note that this metric uses the *maximum period* for a benchmark run, which means any performance disruption due to the OS that would affect real-time predictability is factored into this metric.

Our definition of sustained processor utilization percentage should be used for relative comparisons, rather than as an absolute metric. We have observed percentages approaching 100%—a situation that is seemingly impossible, since we use the maximum Mflops/s rating for the machine in the calculation. An explanation for these observations may be found in the formula used to calculate the number of floating point operations for an FFT: $10n^2 \log_2 n$. The actual number of floating point operations may be less. It is conceivable that the built-in FFT routines have been optimized to limit the number of floating point operations at the expense of more complicated code relying on faster integer operations. Consequently, the true sustained processor utilization percentages are most likely lower than what we report below.

We present graphs of sustained processing utilization in two different formats:

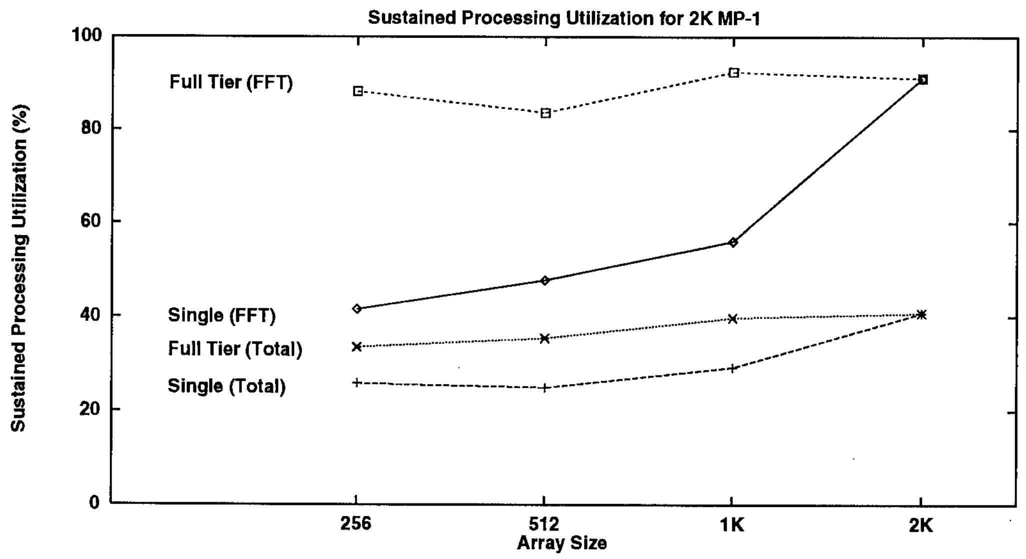
1. Sustained processing utilization as a function of input array size for the 2K processor MasPar MPPs (Figure 14),

2. Sustained processing utilization for various machine sizes (Figure 15).

Each of the figures has two separate graphs—for both the MP-1 and MP-2 machines.

In Figure 14, we present families of curves in each graph of sustained processor utilization versus input array size for:

- Full tier (FFT)—considers only timing for the FFTs and corner turn for adequate input arrays that a full row is assigned to a single processor,



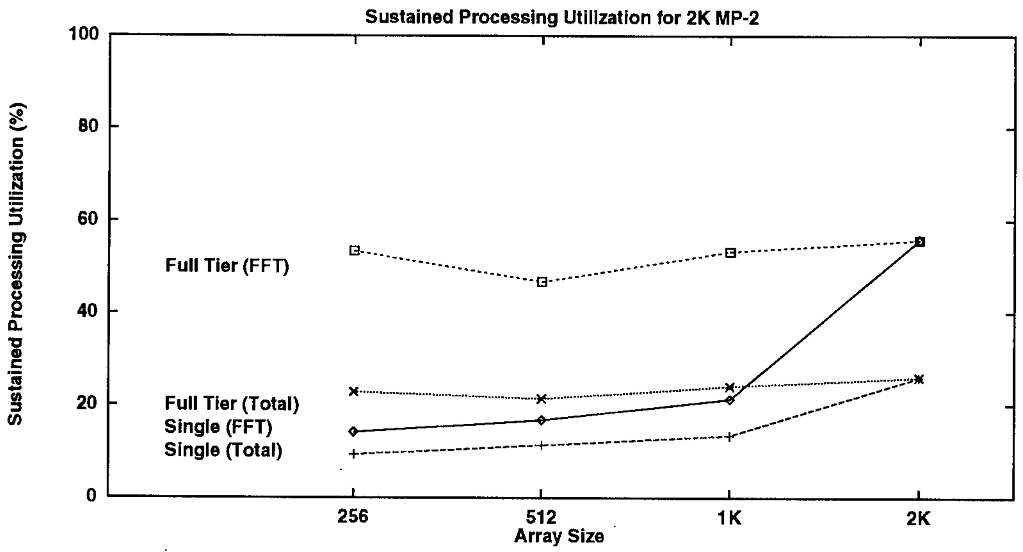
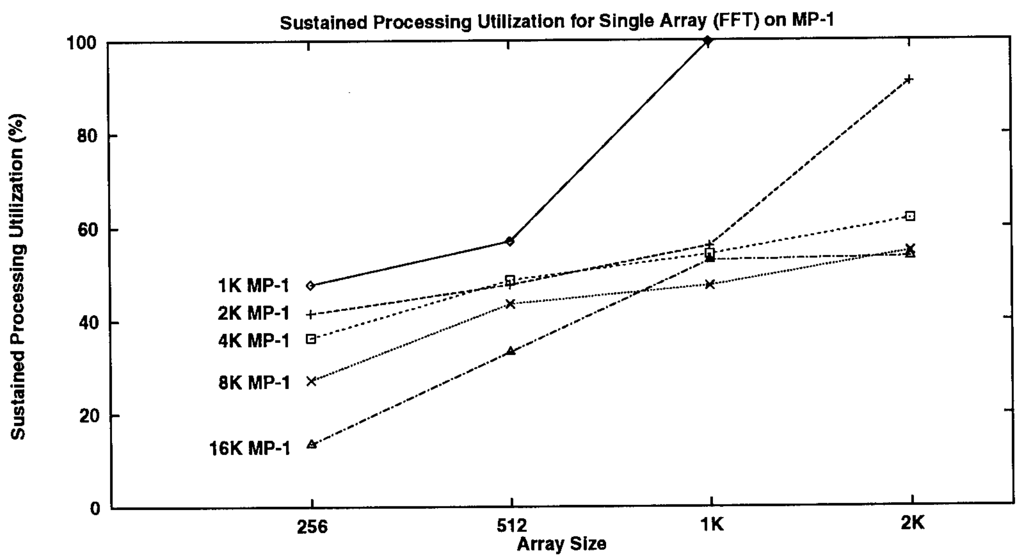


Figure 14. Sustained Processing Utilization for 2K MasPar MPPs

- Full tier (Total)—considers both FFT and I/O timings for adequate input arrays that a full row is assigned to a single processor,
- Single (FFT)—considers only timing for the FFTs and corner turn for a single input array,



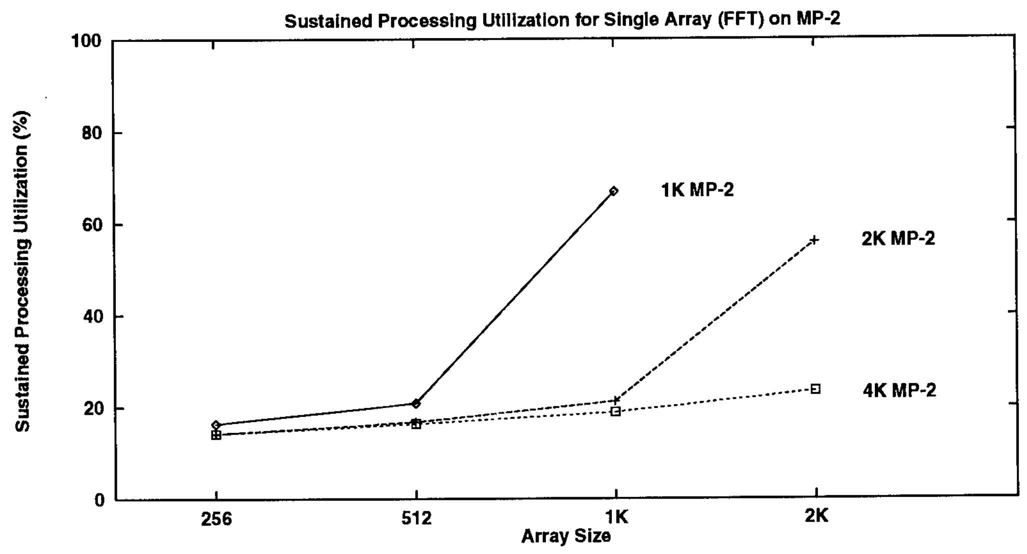


Figure 15. Sustained Processing Utilization for Various Machine Sizes

- Single (Total)—considers both FFT and I/O timings for a single input array.

Sustained processor utilization is significantly better for the full tier case than the single input array case, until such a point that a single input array meets the criteria of also being full tier. This occurs when we examine performance of a 2K input array on the 2K processor machine. In general, sustained processor utilization is monotonically increasing for the single input array case; however, this cannot be said for the full tier case. This is not unexpected, because there is reduced communications as input array size increases for a single input array. Meanwhile, for the full tier case, the amount of communications remains fixed, and only the communications pattern changes for the corner turn.

When comparing the two graphs in Figure 14, we can compare the performance for the MP-1 and MP-2. The sustained processor utilization is much higher for the slower MP-1, because while the MP-2 has faster processors, interprocessor communications capabilities remain the same. The differences in sustained processor utilization is significant for the two machines. Full tier FFT sustained processor utilization is generally greater than 90% for the MP-1; meanwhile, the corresponding curve for the MP-2 shows that sustained processor utilization is less than 60%. As stated above, this disparity in performance is caused by the relationship between computation and interprocessor communications on the two machines.

When comparing sustained processor utilization for FFT-only and total timings, it is evident that we cannot hide the I/O from the simulated data source and to the data sink. Because of the SIMD nature of the MasPar MPPs, there is no way to double buffer data as it is moved from the data source to the PE array and back to the data sink. Sustained processor utilization for floating point operations is greatly reduced when the time for I/O is included in maximum period.

In Figure 15, we present families of curves in each graph of sustained processor utilization for all possible machine sizes with which we tested the `RT_2DFFT` implementation. Again, there are separate graphs for each of the two machines with five separate curves for sustained processor utilization on the MP-1 and three separate curves for the MP-2. We present sustained processor utilization curves only for the FFT-only timings for a single input array. This figure shows that sustained processor utilization is better for smaller machines—an observation that should be obvious because there is reduced interprocessor communications required to perform a two-dimensional FFT when more data is on each processor that performs the one-dimensional FFTs on the rows and columns. There is better sustained processor utilization on the MP-1 than the MP-2, although there is greater variance for small input array sizes on the MP-1 than on the MP-2.

It is instructive to examine those points in Figure 15 that correspond to the minimum machine that meet the one second period constraint for the case when the latency equals the period. For the MP-1, a nearly constant FFT-only utilization percentage between 40% and 50% is maintained as the input array size is increased from 256 to 1K. Maintaining constant resource utilization as the problem size increases is very useful from the standpoint of embedded applications.

This figure also shows that sustained processor utilization follows a trend common with the other scalability metrics in parallel processing—as we increase the number of processors, we must increase the input array size or problem size, otherwise sustained processor utilization can decrease significantly. This is attributed to interprocessor communications overhead. As we spread a constant amount of work over additional processors, there will be additional communications—overhead that is not be encountered on the smaller machine size. In general, there may also be unavoidable sequential processing that does not map to the larger machine.

SECTION 8

CONCLUSIONS

This paper reports on the implementation of the `RT_2DFFT` benchmark developed for the MasPar MP-X series of MPPs. This is the first attempt to extend the real-time embedded benchmarking methodology described in (Games, 1996) to a platform other than the Intel Paragon. Our conclusions are divided into three subsections. In the first subsection, we present our conclusions on the lessons learned relative to the overall real-time embedded benchmarking methodology. We then present our conclusions on the performance of the MasPar `RT_2DFFT` benchmark implementation. In the last subsection, we include a discussion of transitioning our results for the MP-1 and MP-2 machines to MasPar's newest machines, the MP-3 and a *scalable ensemble* processor.

REAL-TIME EMBEDDED BENCHMARK METHODOLOGY

An important motivation in developing the MasPar `RT_2DFFT` benchmark implementation was to evaluate the overall real-time embedded benchmark methodology proposed in (Games, 1996). Our `RT_2DFFT` implementation revealed little yet unseen in previous work on the Intel Paragon (Brown, 1994; Brown, 1995), even though there are significant architecture differences between the MIMD Intel Paragon and the SIMD MasPar MPPs. The MasPar `RT_2DFFT` implementation was simple and straight forward. When MIMD architecture guidelines are inappropriate for the SIMD machine, modifications to the guidelines have been made that maintained the spirit of the `RT_2DFFT` benchmark. In spite of the differences in architectures, each point in the original guidelines have been maintained—with appropriate modifications. These modifications centered around the implementation of the data source and sink and the implementation differences that result from the architecture differences between individual MIMD processing nodes versus a SIMD PE array.

MASPAR `RT_2DFFT` BENCHMARK IMPLEMENTATION

The SIMD architecture of the MasPar MPPs limited the options available when implementing the `RT_2DFFT` benchmark. A SIMD architecture can only process either a single synchronous problem or multiple synchronous problems simultaneously. The combination of the relatively straight forward synchronous, data parallel software architecture of the MasPar with the flexibility of the supplied library functions, contributed to the ease of implementing the MasPar RT_2DFFT benchmark. In particular, software engineering for the MasPar RT_2DFFT benchmark implementation has been greatly simplified by the use of MPL—a C-like programming language. Once, the RT_2DFFT benchmark methodology was conceptually ported to a synchronous, data parallel architecture, the implementation was straightforward.

The MasPar library functions were sufficiently robust and flexible that we developed a single implementation to handle the two latency cases: latency equal to the period, and latency greater than the period. Each latency case has a distinct solution. When latency must equal the period, all PE array resources must be assigned to the task of processing a single input array. The RT_2DFFT benchmark requirement to find the smallest machine able to process the input array within the specified inter-arrival time ensures that the entire PE array participates in the processing. For the latency-greater-than-period case, the most efficient data/processor mapping is to assign an entire row of an input array to each processor by processing sufficient input arrays simultaneously to fill the entire machine. There is no additional processing performance gain achievable by placing multiple input array rows per processor or multiple tiers of input arrays on the machine to fill available memory on the PE array. On the contrary, in some instances the amount of required memory per processor element can be reduced, effectively reducing the size, weight, power consumption, and cost of the computer.

An important feature of the RT_2DFFT benchmark methodology is the ability to examine performance predictability, regardless of the effects of the operating system and other effects beyond the user's control. If there is a clear understanding of predictability, it is possible to use computers with non-real-time operating systems for real-time processing. We found that when using the RT_2DFFT benchmark implementation, it was possible to get very predictable results—as long as all user activity on the PE array was limited to our benchmark. Our results show that much of the measured variability occurred as a side-effect of collecting the timing data that required an operating system call to the UFE processor. We illustrated this in timing tests when we compared multiple fifteen minute program runs on the MP-1. There was substantially more timing variability with clock insertions for each of five program sections than when performing only a single global clock insertion. Because parallel programs running on the MasPar MPP can be run with essentially *no* interaction with the UFE, the MP-X architecture can provide excellent predictability and consequently, excellent real-time performance.

Our RT_2DFFT benchmark implementation tests show that the MasPar MP-1 and MP-2 can process 256 × 256, 512 × 512, and 1K × 1K single precision complex input arrays within the one second inter-arrival time requirement. The smallest machine capable of meeting the timing specification for the latency-equal-period case varied from 1K to 8K for the MP-1 while a 1K MP-2 was able to process each of the three input array sizes successfully. We were in fact unable to process a 2K × 2K input array in the required inter-arrival time on either of the MasPar machines that we used to test our RT_2DFFT implementation; however, we anticipate that we should be able to process a 2K × 2K input array on an 8K processor MP-2 within the one second inter-arrival time requirement. We have shown that the implementation for the latency-greater-than-period case has reduced the measured periods, and in one instance, we could have reduced the machine size by accepting increased latency.

We identified constraints on maximum problem size for two-dimensional applications with single precision complex data array elements and with output streams as great as the input array stream. Some constraints are due to the hardware architecture and some constraints are due to the computational capabilities. Constraints on maximum PE memory impose a potential limitation on input array sizes of 8K × 8K rows and columns. Furthermore, maximum IORAM memory constraints reduce the maximum input array size to 4K × 4K rows and columns, and maximum I/O subsystem bandwidth constraints further reduce the maximum input array size to only 2K × 2K rows and columns.

Processing input arrays larger than 2K × 2K would not be possible on either the MP-1 or MP-2 even if I/O subsystem bandwidth limitations are overcome, assuming that we maintain the one second period requirement. We found that as the number of rows in the input array are doubled, the number of processors required to meet a constant timing requirement must increase by at least a factor of four. While this in not unexpected when processing two-dimensional input arrays for a two-dimensional FFT application, present MasPar processors are limited to only 16K processors and consequently growth to larger input arrays would be limited by the maximum size of the PE array. Given the trends apparent in the data, at least a 32K processor machine or a system speedup of a factor of four would be required to handle a 4K × 4K input array.

EXTENDING THE MASPAR RT_2DFFT BENCHMARK IMPLEMENTATION

An important concern when developing the RT_2DFFT benchmark is the extensibility of the benchmark to other applications and to other hardware, including new versions of an architecture. We design and implement an algorithm on an existing architecture; however, we must predict performance for future architectures (Koester, 1995). Performance predictions for future architectures will help determine whether or not to port larger applications to existing hardware or to wait and look for performance improvements to make implementation of another application feasible.

Our RT_2DFFT implementation was run on available MasPar MP-1 and MP-2 MPPs, and we must consider the extensibility of this work to future SIMD MPPS from MasPar. The basic MasPar MPP architecture has been relatively unchanged since its introduction in 1990. The unusually long lifespan of the MasPar MP-X series MPP technology is attributable to architecture scalability, leveraging the best computer science technologies, adherence to standards, and cost effective manufacturing techniques. MasPar is currently developing the MP-3; although, there is little non-proprietary performance information available on this new machine. It has been announced that the MP-3 parallel processing array architecture will be similar to the previous MasPar MPPs, but will be designed to be faster and eventually will be packaged for embedded applications in conjunction with the Litton Corporation.

The MP-2 series saw increased processor performance by a factor of four over the MP-1, although router-based network performance was not enhanced in the MP-2. Our performance results illustrated that we were unable to get full benefit of the increased processor performance, because communications capabilities did not scale with processor performance. For parallel algorithm performance to scale with respect to processor performance increases, interprocessor communications performance must increase at least as much as processor performance. History has shown that increases in processor performance have been easier to achieve than increases in communications performance.

If both processor performance and the router-based communications network were to see an increase in performance similar to the processor performance increase between the MP-1 and MP-2 architectures—a factor of four—we would expect significant decreases in the size of machines required to perform the RT_2DFFT benchmark. Given such a significant performance increase, even more complicated applications like SAR processing could be performed within the specified input array inter-arrival time. In addition, if the new machine had parallel external access to the I/O subsystem, then the MasPar MP-3 could process larger input arrays, with array size limitations imposed by memory limitations and not I/O bandwidth. It might even be possible to perform two-dimensional FFTs on 8K $\times$ 8K input arrays.

DARPA has also funded MasPar to examine a *scalable ensemble*—multiple MasPar PE arrays interconnected by extensions to the router-based communications system. A scalable ensemble MPP would be a hybrid processor that combined MIMD and SIMD architectures. The MasPar `RT_2DFFT` benchmark methodology and implementation would change significantly for the scalable ensemble architecture. For a cluster of MasPar PE arrays interconnected by the MasPar global router, separate programs could be implemented on each of the individual PE arrays and data pipelined between the PE arrays. As a result, the new `RT_2DFFT` implementation for this ensemble architecture would combine features of both the `RT_2DFFT` MIMD and SIMD implementations.

The COTS MasPar architecture has strong potential for use as a real-time processor; although, the presently available hardware only can meet the `RT_2DFFT` benchmark requirements for a limited number of input array sizes due to system performance limitations and I/O bottlenecks. Additional work on the MasPar `RT_2DFFT` benchmark would be warranted for the MasPar MP-3 or the new scalable ensemble architecture, if interprocessor communications performance is improved commensurate with processor performance and if parallel external I/O capabilities are developed. As a result, performance would scale and larger input arrays or more complicated problems can be addressed in real-time on these new MasPar architectures.

We emphasize that all the results apply to the `RT_2DFFT` benchmark specification given in appendix A. This benchmark specification maintains a one second inter-arrival period independent of input array size. As such, it represents a substantial real-time test of the underlying hardware and system software as the input array size is increased. Any actual application that would require such two-dimensional FFT processing, e.g., SAR, could have a longer inter-arrival time and specific latency requirements. The parametric benchmarking techniques and infrastructure described in this report could be easily adapted to assess the suitability of the MasPar MP-X architecture for a particular target application once the actual real-time requirements are specified.

LIST OF REFERENCES

Blank, T., January 1990, "The MasPar MP-1 Architecture," *IEEE Computer Architecture*, pages 20–24.

Brown, C. P., M. I. Flanzbaum, R. A. Games, and J. D. Ramsdell, October 1994, *Real-Time Embedded High Performance Computing: Application Benchmarks*, MTR 94B145, MITRE, Bedford, MA.

Brown, C. P., R. A. Games, and J. J. Vaccaro, July 1995, *Real-Time Parallel Software Design Case Study: Implementation of the RASSP SAR Benchmark on the Intel Paragon*, MTR95B95, MITRE, Bedford, MA.

Fox, G., M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, 1988, *Solving Problems on Concurrent Processors*, Prentice Hall.

Fox, G. C., 1994, *Software and Hardware Requirements for Some Applications of Parallel Computing to Industrial Problems*, SCCS 717, The Northeast Parallel Architectures Center (NPAC), Syracuse University, Syracuse NY 13244-4100. Available at
`http://www.npac.syr.edu/techreports/html/0700/abs-0717.html`.

Games, R. A., March 1996, *Benchmarking Methodology for Real-Time Embedded Scalable High Performance Computing*, 96B10, MITRE, Bedford, MA.

Hwang, K. and F. A. Briggs, 1984, *Computer Architecture and Parallel Processing*, McGraw Hill.

Koester, D. P., 1994, *SuperComputing-94 — Networking Technologies Summary*, SCCS 708, The Northeast Parallel Architectures Center (NPAC), Syracuse University, Syracuse NY 13244-4100. Available at
`http://www.npac.syr.edu/techreports/html/0700/abs-0708.html`.

____________, October 1995, "Parallel Block-Diagonal-Bordered Sparse Linear Solvers for Power System Applications," Ph.D. thesis, Syracuse University, Syracuse University, Syracuse NY 13244-4100.

MasPar Computer Corp., November 1992, *The Design of the MasPar MP-2: A Cost Effective Massively Parallel Computer*, MP/P-11.92, MasPar Computer Corp., 749 North Mary Avenue, Sunnyvale CA 94086.

MasPar Data Display Library (MPDDL) Reference Manual, July 1992, 749 North Mary Avenue, Sunnyvale CA 94086: MasPar Computer Corp.

MasPar Image Processing Library (MPIPL) Reference Manual, July 1992, 749 North Mary Avenue, Sunnyvale CA 94086: MasPar Computer Corp.

MasPar Mathematics Library (MPIPL) Reference Manual, July 1992, 749 North Mary Avenue, Sunnyvale CA 94086: MasPar Computer Corp.

MasPar Parallel Application Language (MPL), July 1993, 749 North Mary Avenue, Sunnyvale CA 94086: MasPar Computer Corp.

MasPar HiPPI I/O Controller, June 1994, 749 North Mary Avenue, Sunnyvale CA 94086: MasPar Computer Corp.

Pickard, K. T., 1995. Personal communication.

Press, W. H., B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, 1986, *Numerical Recipes*, New York, NY: Cambridge University Press.

Quinn, M. J., 1987, *Designing Efficient Algorithms for Parallel Computers*, McGraw Hill.

Stone, H. S., 1987, *High Performance Computer Architecture, 2nd Ed.*, Addison-Wesley.

Zuerndoerfer, B. and G. A. Shaw, 1994, "SAR Processing for RASSP Application," In *Proceedings of the 1st Annual RASSP Conference*, ARPA, pages 253–268.

APPENDIX A

RT_2DFFT: REAL-TIME SYMMETRIC TWO-DIMENSIONAL FFT BENCHMARK SPECIFICATION

The RT_2DFFT benchmark was proposed in (Games, 1996) as a test of a proposed benchmarking methodology for real-time embedded scalable high performance computing. The two-dimensional FFT benchmark is a kernel commonly used in synthetic aperture radar (SAR) processing, among other applications. In this benchmark, the two-dimensional FFT is treated as a compact application to illustrate the proposed real-time benchmarking methodology. The benchmark assesses the parallel processor's ability to deliver high sustained utilization on an FFT. It also assesses the performance impact of a distributed corner turn global communications operation. This is particularly relevant as the problem size increases and more distributed processing resources are required to meet the real-time requirement (as in higher resolution and/or wider area SARs). The RT_2DFFT benchmark specification consists of a benchmark specification, a timing specification, a scalability study specification, and implementation guidelines.

Notation

In the following, the complex numbers are denoted by $\mathbf{C}$. The set of vectors of size n with entries in a set X is denoted by X^n; the set of two-dimensional input arrays of size $m \times n$ is denoted by $X^{m \times n}$. For an input array $A = \{a_{ij}\} \in X^{m \times n}$, the notation $a_{i,*}$ denotes the ith row of A and $a_{*,j}$ denotes the jth column of A.

Functional Specification

We assume the availability of an FFT algorithm, $y = \mathrm{FFT}(x, \mathit{FFTsize})$, where x and y are complex vectors of size $\mathit{FFTsize}$ and y is the discrete Fourier transform (DFT) of x. See for instance (Press, 1986), page 381. For n a positive integer, the RT_2DFFT benchmark applies the FFT to the rows of an $n \times n$ input matrix A to form the intermediate matrix Y. The output matrix Z is then formed by applying the FFT to the columns of Y. The functional specification of the RT_2DFFT benchmark is given in Figure A-1. Single precision floating point processing is assumed.

$y = $ RT_2DFFT(A, n)

Input: n a positive integer

Input: $A = \{a_{ij}\} \in \mathbf{C}^{n \times n}$

Auxiliary: $Y = \{y_{ij}\} \in \mathbf{C}^{n \times n}$

Output: $Z = \{z_{ij}\} \in \mathbf{C}^{n \times n}$

Algorithm:

```
for $i \in \{0, 1, \ldots, n-1\}, y_{i,*} = \mathrm{FFT}(a_{i,*}, n)$
for $j \in \{0, 1, \ldots, n-1\}, z_{*,j} = \mathrm{FFT}(y_{*,j}, n)$
```

Figure A-1. The RT_2DFFT Benchmark Functional Specification

Timing Specification

The timing specification of the RT_2DFFT benchmark is given in terms of a periodic sequence of input matrices $A_1, A_2, A_3, \ldots A_i, \ldots$. Two requirements are typical in these periodic applications: the period is the time interval between successive input matrices, and the latency is the length of time required to process a single instance A_i, measured as the interval of time between when the matrix A_i leaves the data source and the corresponding results arrive at the data sink. This benchmark fixes the period at 1 second and considers two separate latency cases:

Case 1: period = latency = 1 second

Case 2: period = 1 second, no constraint on latency

The 1 second period for $n = 1024$ or 2048 corresponds roughly to the computational requirements of the SAR system described in (Zuerndoerfer, 1994). In the RT_2DFFT benchmark the timing specification is fixed for all problem sizes.

Scalability Study Specification

The scalability study for the RT_2DFFT benchmark increases the size of the $n \times n$ input matrix, while the period and latency is kept fixed. The objective is to

Table A-1. Processing Rate and Memory Requirements for the RT_2DFFT Benchmark

n	Mflop/s	Mbytes
256	5.2	0.5
512	23.6	2.0
1,024	104.9	8.0
2,048	461.4	32.0
4,096	2,013.3	128.0
8,196	8,724.2	512.0
16,384	37,581.0	2048.0

determine the minimum machine size that meets the real-time requirement. The values of n to be considered are: 256, 512, 1024, 2048, 4096, 8192, and 16,384. Table A-1 shows the computational throughput requirements expressed in Mflop/s. We assume an FFT of length n requires $5n \log_2 n$ floating point operations, so that the total number of floating point operations for the RT_2DFFT benchmark is $10n^2 \log_2 n$. This requirement is common to both latency cases and is based on the single period of 1 second. Table A-1 also shows the memory requirements in megabytes (Mbytes) to store one copy of the input matrix, assuming 8 bytes per element (single precision floating point complex).

Implementation Guidelines

1. The input matrix must be stored originally on a source node in memory that is not directly associated with the processors that implement the RT_2DFFT benchmark. The matrix must be stored in row major or column major form.

2. When establishing timing performance, the same matrix can be repeatedly input to the processors that implement the RT_2DFFT benchmark (to avoid the need for disk I/O).

3. The results must be output to a sink node and stored in memory not directly associated with the processors that implement the `RT_2DFFT` benchmark. The result matrix must be stored in row major or column major form.

4. The source and sink nodes may be implemented on the same or different processing nodes.

5. The processing latency for a problem instance is measured as follows. A time stamp t_s is calculated at the data source right before the input data for this instance is sent from the source node. A second time stamp t_c is calculated at the data sink right after the corresponding results are received by the sink node. The processing latency for this problem instance is then $(t_c - t_s)$. This requires a synchronized global clock if the source and sink are on physically separated nodes. Period measurements are calculated as the difference of successive values of t_c corresponding to successive problem instances. Latency and period measurements can be calculated off-line from the time stamp data.

6. In the case that the input matrix (and output matrix) does not fit in the memory of a single node, then multiple source and sink nodes are necessary. The time stamp t_s of a problem instance should occur before any data is sent from the multiple sources. The processing of that instant is considered completed when all sink nodes have received all their results.

7. A benchmark run to establish timing performance should last for at least 15 minutes to account for any operating system dropout problems.

8. The following information and statistics should be calculated for a single benchmark run (during a post processing stage):

   a. Histogram of period measurements. Maximum, average, and minimum period. The benchmark is considered valid only if the maximum period observed is less than or equal to the period specification: 1 second. This must be repeatable.

   b. Histogram of latency measurements. Maximum, average, and minimum latency. The benchmark is considered valid only if the maximum latency observed is less than or equal to the latency specification. This must be repeatable.

c. Some small number of initial problem instances can be ignored to eliminate start-up anomalies, if present. If this is done then the number of ignored instances should be stated.

9. Machine size is measured in terms of the number of processing nodes used, not including processing nodes used to implement the source and sink. Standard commercial-off-the-shelf hardware and system software configurations should be used. If a machine supports multiple configurations (for example, different amounts of memory at the processing nodes), then these different configurations must be itemized and benchmarked separately. The size, weight, power, and price of each machine should be determined and expressed as a function of machine size.

10. The maximum period for a benchmark run is used to determine the sustained Mflop/s processing rate: $10n^2 \log_2 n/(\text{maximum period})$. This value should be divided by the theoretical peak processing rate of the size machine used in the processing to determine the sustained processing utilization percentage.

11. The following scalability plots should be generated for each latency case:

    a. Minimum machine size as a function of problem size.

    b. Sustained processing utilization percentage as a function of problem size.

APPENDIX B

MASPAR INDEX-BIT PERMUTATIONS

In this appendix, we discuss a set of MasPar subroutine calls that perform certain internal communications called *index-bit permutations.* These routines allow for a very efficient implementation of the two data rearrangements required by the RT_2DFFT benchmark implementation. One rearrangement is the shift needed to put the data in cut-and-stack form and the other rearrangement is the data transpose used for the corner turn. We will describe the specifics of each of these functions after we give an overview of the philosophy behind the index-bit permutations.

Consider a plural data set where each PE contains a linear array of data of some fixed length, which for simplicity we take to be a power of two. Let 2^k be the length and suppose that there are 2^q PEs. Then every data element in the plural set can be specified by a unique $(q+k)$-bit index:

$$(\overbrace{p_0, p_1, \ldots, p_{q-1}}^{\text{PE bits}}; \overbrace{m_0, m_1, \ldots, m_{k-1}}^{\text{Mem bits}}) \qquad \text{(B-1)}$$

We are not concerned with the actual size of the data element, since the routines can move data elements of any size in byte increments (e.g., in the RT_2DFFT benchmark implementation, a data element is a single precision complex using eight bytes).

The goal of the index-bit permutations is to permute the data elements by simple permutations of the bit representation. That is, we permute the bits in the index given in equation B-1, say for example, by switching p_0 with m_0. The data elements are then themselves permuted consistent with the permutation of their indices. There are several options for permutations of the bit representation. Some permutations only permute PE bits; others interchange PE bits with Mem bits.

Cut-and-Stack

As we mentioned in the MasPar performance section, the fastest method for reading data from the IORAM is to read a contiguous portion of the data onto a PE and then to rearrange the data into the required cut-and-stack form. We illustrate this mapping for a set of four PEs acting on a row of length 32; the data are input as in the left matrix below and must be transformed to the matrix on the right:

$$\begin{matrix} 0 & 8 & 16 & 24 \\ 1 & 9 & 17 & 25 \\ 2 & 10 & 18 & 26 \\ 3 & 11 & 19 & 27 \\ 4 & 12 & 20 & 28 \\ 5 & 13 & 21 & 29 \\ 6 & 14 & 22 & 30 \\ 7 & 15 & 23 & 31 \end{matrix} \quad \longrightarrow \quad \begin{matrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 \\ 20 & 21 & 22 & 23 \\ 24 & 25 & 26 & 27 \\ 28 & 29 & 30 & 31 \end{matrix}$$

In this example there are 3 Mem bits, since the columns are length eight. The number of PE bits could be any $q \geq 2$, but since only 4 PEs are used for a row, only the lowest 2 PE bits are used for the given row. There are then five bits total for the data in the row, say a, b, c, d, e (from lowest to highest). The higher two bits d and e are the PE bits. We need to change

$$(d, e, p_2, \ldots, p_{q-1}; a, b, c) \qquad \text{to} \qquad (a, b, p_2, \ldots, p_{q-1}; c, d, e).$$

If we ignore for the moment the p_i bits, then we see that we must change (d, e, a, b, c) to (a, b, c, d, e). This is just a cyclic shift of (d, e, a, b, c) by three to the right. The p_i bits are fixed and not permuted. The final result is that we perform the same data rearrangement for every row of every two-dimensional input array, because the index of the rows and arrays are expressed by the p_i bits.

There is a specific routine, called `cshift_pm` that is designed to do precisely this mapping. We can perform the rearrangement of the data to cut-and-stack form with just one call to this routine. This transformation is not required when more than one tier of input arrays is processed, since only one PE is used per row (so there are zero PE bits needed for the row index).

Corner Turns

The corner turn of a two-dimensional data array is conceptually similar to the transformation used in the cut-and-stack mapping, but we require two index-bit routines. For the corner turn, we start with data in cut-and-stack and consider bits used to index rows of the input array (r_i), columns of the input array (c_i), and possibly any multiple input arrays (a_i). The data elements are specified by the index

$$(r_0, \ldots, r_{t-1}, c_0, \ldots, c_{s-1}, a_0, \ldots, a_{q-s-t-1}; r_t, \ldots, r_{s-1}),$$

where as always PE bits are given first, t is the number of PE bits used for a single row, and s is the number of bits required for the data array size (i.e., we process $2^s \times 2^s$ data arrays). The corner turn performs a transpose of the data that interchanges r_i and c_i. That is, the final index is given by

$$(c_0, \ldots, c_{t-1}, r_0, \ldots, r_{s-1}, a_0, \ldots, a_{q-s-t-1}; c_t, \ldots, c_{s-1}).$$

The first step to do the corner turn is to swap the Mem bits with the appropriate PE bits. This is done using the swap_pm subroutine and results going from the starting index

$$(r_0, \ldots, r_{t-1}, c_0, \ldots, c_{t-1}, \overbrace{c_t, \ldots, c_{s-1}}, a_0, \ldots, a_{q-s-t-1}; \overbrace{r_t, \ldots, r_{s-1}})$$

to an intermediate index

$$(r_0, \ldots, r_{t-1}, c_0, \ldots, c_{t-1}, \overbrace{r_t, \ldots, r_{s-1}}, a_0, \ldots, a_{q-s-t-1}; \overbrace{c_t, \ldots, c_{s-1}})$$

by swapping the two indicated blocks of bits. The next step is just to exchange the lower order r_i and c_i. This is done using the xchng_pp subroutine and transforms the intermediate index

$$(\overbrace{r_0, \ldots, r_{t-1}}, \overbrace{c_0, \ldots, c_{t-1}}, r_t, \ldots, r_{s-1}, a_0, \ldots, a_{q-s-t-1}; c_t, \ldots, c_{s-1})$$

to

$$(\overbrace{c_0, \ldots, c_{t-1}}, \overbrace{r_0, \ldots, r_{t-1}, r_t}, \ldots, r_{s-1}, a_0, \ldots, a_{q-s-t-1}; c_t, \ldots, c_{s-1})$$

by exchanging the indicated blocks of bits. This last index is in the desired final form. Notice that the a_i bits are fixed and never permuted. This means that each two-dimensional data array is transposed by itself. When more then one tier of arrays is processed, we need to call the two subroutines separately for each tier.

APPENDIX C

MASPAR MP-1 RT_2DFFT BENCHMARK DATA

This appendix gives a complete set of data from the short ten-iteration runs of the RT_2DFFT benchmark implementation on the MP-1 machine. Specifications for the MP-1 machine are given in Table C-1. The MP-1 machine was made available by NPAC at Syracuse University; the runs were made in September, 1995. A separate table is given for each software configured machine size; these are Tables C-2 to C-6. The tables indicate limitations due to the capacity of the IORAM; when there was insufficient IORAM capacity, only the time to perform the 2D FFT processing is reported. All times are in milliseconds and are the maximum time for the ten iterations after ignoring the initial iteration. The range of times for the ten iterations were almost always 10–20 milliseconds.

The headings in these tables are as follows:

n	The size of the input array, that is, $n \times n$.
N_A	The number of input arrays processed simultaneously.
N_T	The number of tiers of input arrays.
Lat	The total time needed to process the input arrays—the latency.
Per	The time to process all of the input arrays divided by the number of input arrays—the period.
Read	The time to read from the IORAM to the PE array. An "—" signifies that there was inadequate IORAM to provide both data source and sink.
FFT$_R$	The time to compute the 1D FFTs on the rows of the input arrays.
CT	The time to perform a corner turn on the data arrays.
FFT$_C$	The time to compute the 1D FFTs on the columns of the data arrays.
Write	The time to write from the PE array to the IORAM. An "—" signifies that there was inadequate IORAM to provide both data source and sink.

Table C-1. Specifications of the MP-1 Machine

	MP-1
Front-End	DECstation
FE OS	Ultrix V4.3
PE Array	16K = 128 × 128
ACU IMEM	1 Mbytes
ACU CMEM	1 Mbytes
PE MEM	64 Kbytes
IORAM Size	32 Mbytes

Table C-2. 1K MasPar MP-1 Empirical Data

n	N_A	N_T	Lat	Per	Read	FFT$_R$	CT	FFT$_C$	Write
256	1	1	282	282	78	56	31	59	70
	2	1	559	280	161	129	39	133	134
	4†	1	793	198	290	129	55	129	231
	8	2	1579	197	565	250	106	247	451
	16	4	3145	197	1118	486	222	474	887
	32	8	6277	196	2223	958	430	949	1769
512	1	1	1094	1094	312	250	47	254	258
	2†	1	1743	872	563	314	146	313	454
	4	2	3465	866	1117	616	279	618	887
	8	4	6922	866	2223	1212	543	1216	1769
1K	1†	1	3359	3359	1118	617	169	617	887
	2	2	6711	3356	2220	1223	321	1224	1767

† Full tier of input arrays.

Table C-3. 2K MasPar MP-1 Empirical Data

n	N_A	N_T	Lat	Per	Read	FFT_R	CT	FFT_C	Write
256	1	1	126	126	28	32	20	32	23
	2	1	287	144	83	56	36	59	63
	4	1	559	140	161	133	32	126	134
	8†	1	793	99	285	129	59	129	231
	16	2	1587	99	567	242	111	243	454
	32	4	3149	98	1118	484	232	485	888
	64	8	6301	98	2223	949	455	945	1767
512	1	1	593	593	163	133	59	137	137
	2	1	1122	561	317	251	78	254	257
	4†	1	1727	432	565	310	129	313	453
	8	2	3441	430	1118	618	250	614	888
	16	4	6865	429	2223	1215	488	1216	1770
1K	1	1	2341	2341	629	559	126	560	513
	2†	1	3466	1733	1118	618	278	616	886
	4	2	6930	1733	2219	1223	547	1223	1770
2K	1†	1	7539	7539	2305	1485	406	1484	1859

† Full tier of input arrays.

Table C-4. 4K MasPar MP-1 Empirical Data

n	N_A	N_T	Lat	Per	Read	FFT$_R$	CT	FFT$_C$	Write
256	1	1	54	54	12	16	12	20	12
	2	1	133	67	27	31	24	32	24
	4	1	288	72	79	58	39	56	63
	8	1	556	70	165	125	36	129	133
	16†	1	1028	68	412	125	59	125	335
	32	2	2036	64	804	246	113	242	665
	64	4	4063	63	1599	485	232	484	1319
	128	8	3376‡	3376‡	—	950	454	947	—
512	1	1	302	302	78	63	36	62	67
	2	1	607	302	160	134	70	137	138
	4	1	1349	337	438	250	79	255	371
	8†	1	2180	273	806	314	133	314	664
	16	2	4353	272	1601	614	250	614	1321
	32	4	2941‡	2941‡	—	1215	485	1212	—
1K	1	1	1411	1411	439	255	134	255	367
	2	1	2829	1415	867	556	160	556	730
	4†	1	4379	1095	1598	617	278	617	1321
	8	2	3008‡	3008‡	—	1223	544	1222	—
2K	1	1	5641	5641	1739	1109	276	1105	1453
	2†	1	3523‡	3523‡	—	1489	548	1486	—

† Full tier of input arrays.
‡ Two-dimensional FFT timing only.

Table C-5. 8K MasPar MP-1 Empirical Data

n	N_A	N_T	Lat	Per	Read	FFT_R	CT	FFT_C	Write
256	1	1	51	51	16	12	8	12	12
	2	1	55	28	12	16	12	19	8
	4	1	133	33	28	35	24	31	24
	8	1	288	36	82	56	39	55	63
	16	1	602	38	185	133	39	133	156
	32†	1	1031	32	406	129	59	129	337
	64	2	2047	32	806	239	134	243	661
	128	4	1211‡	1211‡	—	481	250	484	—
	256	8	2408‡	2408‡	—	949	489	949	—
512	1	1	134	134	27	36	19	35	25
	2	1	314	157	83	63	43	63	63
	4	1	645	161	191	129	67	136	160
	8	1	1341	168	433	251	71	247	369
	16†	1	2187	137	806	314	134	314	666
	32	2	1485‡	1485‡	—	614	255	615	—
	64	4	2949‡	2949‡	—	1219	497	1215	—
1K	1	1	684	684	192	145	72	150	160
	2	1	1418	709	438	254	149	254	367
	4	1	2828	707	863	558	161	556	730
	8†	1	1481‡	1481‡	—	617	266	617	—
	16	2	2976‡	2976‡	—	1222	517	1223	—
2K	1	1	2969	2969	872	572	260	568	729
	2	1	2516‡	2516‡	—	1110	313	1107	—
	4†	1	3524‡	3524‡	—	1485	548	1483	—

† Full tier of input arrays.

‡ Two-dimensional FFT timing only.

Table C-6. 16K MasPar MP-1 Empirical Data

n	N_A	N_T	Lat	Per	Read	FFT$_R$	CT	FFT$_C$	Write
256	1	1	40	40	12	12	8	12	8
	2	1	50	25	12	12	8	12	12
	4	1	60	15	12	16	15	16	12
	8	1	133	17	28	36	23	35	24
	16	1	289	18	79	55	39	55	63
	32	1	605	19	185	133	39	133	152
	64†	1	1031	16	407	125	59	128	337
	128	2	613‡	613‡	—	243	133	243	—
512	1	1	86	86	19	24	12	23	20
	2	1	137	69	24	36	23	35	24
	4	1	312	78	81	62	43	63	65
	8	1	642	80	191	134	70	137	157
	16	1	1345	84	438	251	71	255	371
	32†	1	742‡	742‡	—	315	133	313	—
1K	1	1	313	313	86	63	39	63	70
	2	1	688	344	192	144	83	150	157
	4	1	1418	355	438	253	148	254	367
	8	1	1263‡	1263‡	—	559	161	559	—
	16†	1	1481‡	1481‡	—	617	266	613	—
2K	1	1	1500	1500	445	277	157	282	372
	2	1	1411‡	1411‡	—	571	287	570	—
	4	1	2517‡	2517‡	—	1111	311	1110	—
	8†	1	3520‡	3520‡	—	1488	547	1488	—

† Full tier of input arrays.
‡ Two-dimensional FFT timing only.

APPENDIX D

MASPAR MP-2 RT_2DFFT BENCHMARK DATA

This appendix gives a complete set of data from the short ten-iteration runs of the RT_2DFFT benchmark implementation on the MP-2 machine. Specifications for the MP-2 machine are given in Table D-1. The MP-2 machine was made available by the MasPar Computing Corporation; the runs were made in September, 1995. A separate table is given for each software configured machine size; these are Tables D-2 to D-4. All times are in milliseconds and are the maximum time for the ten iterations after ignoring the initial iteration. The range of times for the ten iterations were almost always 1–6 milliseconds.

The headings in the tables are defined in appendix C.

Table D-1. Specifications of the MP-2 Machine

	MP-2
Front-End	Alpha Workstation
FE OS	Digital UNIX (OSF1)
PE Array	$4K = 64 \times 64$
ACU IMEM	4 Mbytes
ACU CMEM	512 Kbytes
PE MEM	64 Kbytes
IORAM Size	128 Mbytes

Table D-2. 1K MasPar MP-2 Empirical Data

n	N_A	N_T	Lat	Per	Read	FFT$_R$	CT	FFT$_C$	Write
256	1	1	119	119	22	29	22	29	18
	2	1	231	116	43	64	31	64	35
	4†	1	219	55	72	27	41	27	58
	8	2	436	55	140	55	78	55	113
	16	4	866	54	278	111	155	111	228
	32	8	1728	54	553	217	307	216	451
512	1	1	427	427	86	123	35	124	72
	2†	1	510	255	139	77	111	78	113
	4	2	1015	254	278	153	216	151	227
	8	4	2026	253	553	301	433	300	450
1K	1†	1	910	908	289	136	120	136	240
	2	2	1816	908	579	268	236	269	474

† Full tier of input arrays.

Table D-3. 2K MasPar MP-2 Empirical Data

n	N_A	N_T	Lat	Per	Read	FFT$_R$	CT	FFT$_C$	Write
256	1	1	67	67	13	17	12	17	11
	2	1	123	62	22	29	28	29	18
	4	1	225	56	43	64	26	64	35
	$8^\dagger$	1	221	28	72	27	44	27	59
	16	2	439	27	140	54	81	54	114
	32	4	872	27	278	110	162	111	228
	64	8	1743	27	553	216	321	216	450
512	1	1	249	249	46	63	50	63	36
	2	1	451	226	85	121	61	122	73
	$4^\dagger$	1	539	135	163	79	94	79	137
	8	2	1072	134	328	152	185	152	269
	16	4	2138	134	650	302	363	302	533
1K	1	1	964	964	195	259	94	261	165
	$2^\dagger$	1	1071	536	328	137	216	138	269
	4	2	2137	534	649	269	427	268	532
2K	$1^\dagger$	1	2202	2202	650	357	318	357	532

† Full tier of input arrays.

Table D-4. 4K MasPar MP-2 Empirical Data

n	N_A	N_T	Lat	Per	Read	FFT_R	CT	FFT_C	Write
256	1	1	31	31	5	8	7	8	5
	2	1	73	37	13	17	17	17	11
	4	1	123	31	22	29	27	29	18
	8	1	243	30	53	63	26	64	43
	16†	1	251	16	89	27	42	27	74
	32	2	500	16	174	54	81	56	142
	64	4	999	16	344	111	160	110	282
	128	8	1991	16	687	215	321	215	561
512	1	1	131	131	23	32	26	32	20
	2	1	272	136	54	64	55	63	46
	4	1	483	121	101	122	60	123	087
	8†	1	608	76	202	78	93	79	170
	16	2	1213	76	401	152	184	152	333
	32	4	2419	76	799	303	363	301	664
1K	1	1	525	525	105	123	102	121	87
	2	1	1061	531	232	261	120	261	199
	4†	1	1212	303	401	137	216	136	334
	8	2	2422	303	799	269	427	269	665
2K	1	1	2079	2079	466	503	219	503	398
	2†	1	2589	1295	799	357	425	357	665

† Full tier of input arrays.

GLOSSARY

ACU	Array Control Unit
ATM	Asynchronous Transfer Mode
C^2	command and control
COTS	commercial off-the-shelf
DARPA	Defense Advanced Research Projects Agency
DEC	Digital Equipment Corporation
DFT	discrete Fourier transform
FFT	fast Fourier transform
HiPPI	High Performance Parallel Interface
I/O	input/output
ITO	Information Technology Office
Mflop/s	millions of instructions per second
MIMD	multiple-instruction stream, multiple-data stream
MIPS	million of instructions per second
MOIE	Mission Oriented Investigation and Experimentation
MPL	MasPar Parallel Application Language
MPIPL	MasPar image processing library
MPML	MasPar mathematics library
MPP	massively parallel processor
NPAC	Northeast Parallel Architectures Center
OS	operating system
PE	processor element
RISC	reduced instruction set computer
RPC	Remote Procedure Call
SAR	synthetic aperture radar
SIMD	single-instruction stream, single-data stream
THAAD	Theater High Altitude Area Defense
TMD-GBR	Theater Missile Defense Ground Based Radar
UFE	UNIX front-end
VLSI	very large scale integration
VME	virtual memory expansion

☆U.S. GOVERNMENT PRINTING OFFICE: 1996-509-127-47067